02 P 14682

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
6 September 2002 (06.09.2002)

PCT

(10) International Publication Number
WO 02/068899 A2

(51) International Patent Classification[7]: G01B

(21) International Application Number: PCT/US02/05341

(22) International Filing Date: 20 February 2002 (20.02.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/792,650 23 February 2001 (23.02.2001) US

(71) Applicant: INVISION TECHNOLOGIES, INC. [US/US]; 7151 Gateway Boulevard, Newark, CA 94560 (US).

(72) Inventor: GARMS, III, Walter, I.; 2879 Louis Road, Palo Alto, CA 94304 (US).

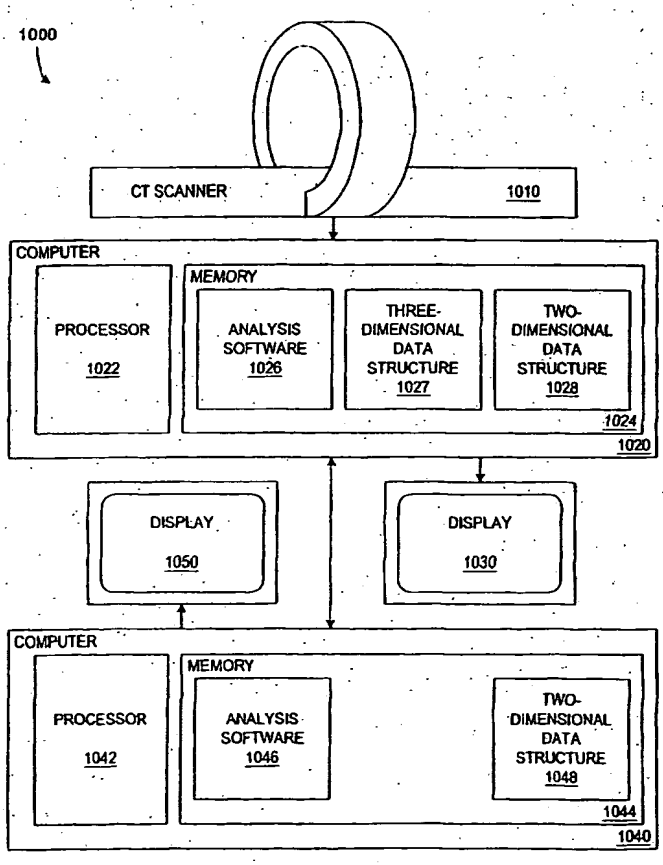
(74) Agents: PARSONS, James, E. et al.; Skjerven Morrill Macpherson LLP, 25 Metro Drive, Suite 700, San Jose, CA 95110 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent

[Continued on next page]

(54) Title: LOG EVALUATION USING CYLINDRICAL PROJECTIONS



(57) Abstract: A three-dimensional density distribution of a log is reduced to a two-dimensional structure that provides a convenient image or visualization of defects like knots or voids in a log, to facilitate grading and/or optimization of a sawing strategy for the log. The two-dimensional data structure is based on cylindrical or modified cylindrical coordinates Z and θ. To provide a more compact identification of defects, modified cylindrical coordinates use a Z-axis that follows the growth center in the log and determines data points by evaluating properties of the log along rays at an upward angle corresponding to limbs in a tree. A process for identifying the growth center at any distance Z along the length of the log examines or accumulates the gradient of density along lines through a cross-section of the log. Manual grading and sawing optimization can employ viewing of an image based on the two-dimensional data structure with superimposed marks indicating the boundaries of faces cut from the log. Automated grading and sawing optimization employs the two-dimensional data structure to reduce processing time when compared to processes that manipulate three-dimensional data structures.

WO 02/068899 A2

(BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

LOG EVALUATION USING CYLINDRICAL PROJECTIONS

BACKGROUND

Logs are non-standard commodities that contain defects such as knots and cracks that significantly affect the value of boards cut from the log. Accordingly, when buying logs, a grader evaluates each log to determine the extent and locations of defects in the log. A sawyer, when cutting the log, attempts to select a sawing strategy that provides the highest value for the lumber cut from the log. These graders and sawyers typically depend on the external appearance of a log when grading or evaluating the log, but viewing the exterior of a log gives the grader or sawyer an imprecise indication of the quality of a log and often fails to indicate internal defects. This results in economic inefficiency because logs are inaccurately graded or cut.

Log scanning devices have been developed to improve the evaluation of logs and the optimization of sawing strategies. For example, U.S. patent 5,394,342 describes a scanning system that applies circumferentially spaced traverse scans along the length of a log to provide longitudinal density data. A paper by D. Schmoldt, entitled "CT Imaging, Data Reduction, and Visualization of Hardwood Logs," Hardwood Symposium Proceedings, (May 1996) and a paper D. Schmoldt et al. entitled "Nondestructive Evaluation of Hardwood Logs: CT Scanning, Machine Vision and Data Utilization," Nondestr. Test Eval., Vol. 15 (1999) describe CT (computer tomography) scanning of logs and evaluation of data resulting from CT scanning of a log.

The three-dimensional data structures and the large amount of data that scanners generate for a log are often difficult to use. For example, data indicating a three-dimensional density distribution for a log is difficult for a grader or sawyer to visualize, and computer manipulation of the large amount of data requires significant processing power or time. Efforts are continuing to improve the methods for quantifying the properties of logs, visualizing the data associated with the logs, and optimizing log grading and sawing strategies based on such data.

SUMMARY

In accordance with an aspect of the invention, a three-dimensional density distribution for a log is processed to provide a convenient image or visualization of defects like knots or voids in a log and thereby facilitate grading and optimization of a sawing strategy for a log. The data representing such images has a two-dimensional structure and

contains considerably less data than a three-dimensional density distribution. Accordingly, manipulation of data structures in accordance with the invention requires less processing power or time in automated processes for grading or optimization of a sawing strategy. Other aspects of the invention provide processes for grading and optimizing the sawing strategies based on the two-dimensional data structures.

In one embodiment, a visualization of a log provides a two-dimensional image where intensities (or colors) in the image correspond to a property evaluated along at least portions of cylindrical projections extending from the center of the log. Exemplary properties shown in the image include but are not limited to the average density along a projection, a minimum or maximum density along a projection, existence of a steep change in density along a projection, and flags indicating the presence or absence of defects along a projection. A central axis from which the projections extend can be straight or can follow the center of growth rings in the log, and the projections can be perpendicular to the axis or extend at an upward angle characteristic of the branches in a tree. A viewer of this image can quickly identify an orientation for the log that presents the fewest defects to a first cut of a saw blade.

One exemplary embodiment of the invention is a data structure for describing a log. The data structure includes a two-dimensional array of data values. Each data value corresponds to values of a coordinate Z and a second coordinate θ. The coordinate Z indicates distance along the log, and the coordinate θ indicates an angle around the log. Each data value indicates a property of the log that is evaluated along a ray that originates at a center point corresponding to the value of the coordinate value Z and extends in a direction corresponding to the coordinate value θ. The center point at each value of the coordinate Z is typically the growth center of the log at that Z-coordinate value but alternatively can be the geometric center of the log or of core wood in the log. The ray evaluated to determine a data value can be perpendicular to the length of the log or directed at an upward angle along the log. The upward angle typically depends on a growth direction characteristic of tree limbs in the species of tree that produced the log or is determined independently for each log. Optionally, each data value indicates a property of the log that is evaluated in a range of distances along the ray that originates at the center point. The range can be limited to exclude core wood in the center of log and or bark on the outside of the log. In one specific embodiment, each data value indicates presence or

absence of a defect corresponding to the coordinates of the data value and may further indicate the depth or location along the ray for any defect on the ray.

Another embodiment of the invention is a method for generating a description of a log. For a set of locations along the length of the log, the method finds a center point (e.g., a geometric or growth center) of the log. Each ray in a set of rays that extend from the center points is evaluated. In particular, the method evaluates a property of the log along each ray to generate a data value corresponding to Z and θ coordinates identifying the direction of the ray. Typically, the evaluated property is density along the ray, and the evaluation determines whether there is a defect along the ray. A CT scan of the log can generate a three-dimensional data structure that provides the densities evaluated along the rays. A two-dimensional data structure that describes the log includes the data values, which are positioned in the two-dimensional data structure according to their respective coordinates.

Another embodiment of the invention is a system for evaluating logs. The system includes program code that is computer executable for manipulating a data structure such as described in the preceding paragraph. Generally, the system further includes a display device and a processor capable of executing the program code. In executing the program code, the processor controls display of an image on the display device. The image includes pixels that correspond to the data values of the data structure and have shades defined by the respective data values. The system can further superimpose marks on the image to indicate boundaries of one or more faces of the log that results from sawing the log. The marks can be shifted relative to the image identify an optimal orientation for sawing the log that minimizes the defects present between the boundary marks, i.e., in the cut faces of the log.

Another embodiment of the invention is method for grading a log. The grading method uses a two-dimensional defect structure or data array as described above that includes data values indexed by cylindrical or modified cylindrical coordinates. The grading method evaluates the two-dimensional array to determine sizes of blocks that are free of data values indicating defects. Each defect-free block contributes to a grade value of the log, according to the size of the block. Evaluation of the two-dimensional array can be performed manually, e.g., by a grader viewing an image having pixels shaded according to corresponding data values. The grader easily and quickly recognizes the sizes of the blocks through evaluating areas of the image having a shade indicating an absence of

defects, and the grader can assign the grade to the log based on the viewing of the image. To aid the grader, superimposed marks in the image indicate boundaries of one or more board faces that results from a sawing strategy for the log, and the grader can shift the image relative to the marks to minimizes defects within the one or more board faces. The manual process can also select an orientation for sawing of the log according to positions of the marks that minimize defects in the board faces.

The grading process can also be automated through a computer program that manipulates the two-dimensional array. One exemplary computer program includes: (a) determining a number N of data values that are consecutive in a direction of the coordinate θ and correspond to a desired width of defect-free wood; (b) scanning the two-dimensional array in the direction of the coordinate θ until identifying N consecutive data values that indicate absence of a defect; (c) scanning the two-dimensional array in a direction of the coordinate Z to determine a size of a block that is defect free; (d) increasing the grade value for the log by an amount corresponding to the size of the defect-free block; (e) repeating steps (b), (c), and (d) to account for all defect-free blocks in the two-dimensional data structure.

Another embodiment of the invention is an automated grading process that accounts for the sawing strategy and the optimal log orientation for the sawing strategy. An exemplary embodiment of this grading method includes: (a) creating a two-dimensional data or defect structure of a type described above; (b) selecting a sawing strategy for the log; (c) selecting an orientation of the log for the sawing strategy; (d) identifying a sub-array of the two-dimensional array, the sub-array corresponding to a face of the log resulting from the sawing strategy and the orientation; (e) evaluating the sub-array to determine sizes of blocks in the sub-array, that are free of data values indicating defects; (f) assigning a first grade value to the face according to the block sizes; (g) repeating steps (d), (e), and (f) for one or more faces of the log resulting from the sawing strategy and the orientation; (h) combining the grade values of the faces to generate a grade value for the orientation; and (i) repeating steps (c) to (h) for one or more additional orientations of the log; (j) assigning a grade value to the log based on a best of the grade values for the orientations.

Yet another embodiment of the invention is method for identifying a growth center of a log. The method determines an accumulated absolute value of a gradient of the density, or the number of zero crossings of the gradient, along lines through a cross-

section of the log and identifies two crossing lines that have the largest accumulated values or the greatest numbers of zero crossings. The growth center is at the intersection of the crossing lines. Generally two sets of lines are considered, wherein the lines in the first set are perpendicular to the lines in the second set. To reduce the number of lines evaluated, the method can determine a geometric center of the log and select a small central area containing the geometric center of the log. The lines evaluated are limited to those passing through the central area. In accordance with a further aspect of the invention, considering only densities for points inside the central area when determining the accumulated absolute values of the gradient of the density reduces the complexity of the calculation.

These and other embodiments of the invention will be better understood in light of the detailed description below when taken with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates Cartesian and modified cylindrical representations of a log.

Fig. 2 illustrates a relationship between a portion of a log and an image or defect structure of the log in accordance with and embodiment of the invention.

Fig. 3A is a flow diagram of a process for determining a defect structure of a log.

Fig. 3B is illustrates density variations in a cross-section of a log as used to identify the center of growth rings in the log.

Fig. 4 is a flow diagram of a process employing a modified cylindrical representation to identify the locations of defects in a log.

Fig. 5 is a flow diagram of ray tracing process used in creating a defect structure in accordance with an embodiment of the invention.

Fig. 6 is a flow diagram of a process of identifying defects that may be hidden by variations in the good wood in a log.

Figs. 7A and 7B illustrate a log grading process in accordance with and embodiment of the invention.

Figs. 8A and 8B illustrate a sawing strategy and a method in accordance with the invention for optimizing yield of clear wood from a log.

Fig. 9 illustrates a log grading practice that in accordance with an aspect of the invention accounts for a sawing strategy and optimization of log orientation when determining a grade for a log.

Fig. 10 is a block diagram of a log analysis system in accordance with an embodiment of the invention.

Use of the same reference symbols in different figures indicates similar or identical items.

DETAILED DESCRIPTION

In accordance with an aspect of the invention, a process for grading logs and optimizing sawing strategies for logs is based on computer tomography data and modified cylindrical projections used to represent the log in two-dimensional images and data structures. This two-dimensional representation facilitates defect identification, dramatically reduces the amount of data required to describe the defect structure of a log, and allows a person or a computer to quickly and efficiently grade a log or optimize a sawing strategy for the log.

Computer tomography (CT) scanning generally provides a three-dimensional data structure. Each value in the data structure corresponds to a small volume (or voxel) and indicates a density or an absorption coefficient for the voxel. Each voxel further has three Cartesian coordinates Xc, Yc, and Zc that indicate the relative locations of the voxels. Known CT scanners can analyze logs and provide three-dimensional data structures representing logs. The amount of data in a three-dimensional data structure depends on the size of the log and resolution required for useful evaluation of the log. Depending on the resolution, a single 4-meter log, for example, might require 800 MB of data in the standard three-dimensional format. This amount of data is time consuming to process. Additionally, the three-dimensional nature of the data makes the data difficult to visualize, store, or transmit.

In accordance with an aspect of the invention, a representation of the defects or other structures in a log uses a modified cylindrical coordinate system. Fig. 1 illustrates a log 100 and a Cartesian coordinate system defining coordinates Xc, Yc, and Zc for each point or voxel in the log. The Zc coordinate represents the distance along the length of the log (up, in a standing tree), and the Xc and Yc coordinates define a location in planes perpendicular to the Zc-axis. A set of CT cross sectional images taken at intervals along the length of the log provide data points (e.g., density values) corresponding to Xc-Yc planes at discrete values of the Zc coordinate. As mentioned above, this three-dimensional system can include a large amount of data, making any sort of optimization and grading process complex and time consuming.

One of the most important aspects of tree structure and the defect structure in a log is the center 125 of the annual growth rings 120. Generally, the center 125 of the annual rings 120, sometimes referred to herein as growth center 125 can be a significant distance from the geometric center of a cross-section of a tree or log, and the position of growth center 125 changes, albeit slowly, with the Zc coordinate. Most significant cracks in a typical log pass through the center of the growth rings (e.g., growth center 125). Knots 110, which are the portions of tree limbs in the tree trunk or log, generally extend upward and radially away from growth center 125. Imbedded knots, which commonly result when the lower branches of a tree break off and the tree grows clear wood over the previous location of the branch, also extend upward and radially away from growth center 125. Imbedded knots may leave little or no visible signs on the surface of a log, but knowledge of the existence and extent of imbedded knots is critical to the sawyer or log grader.

In accordance with an aspect of the invention, an efficient two-dimensional representation or image of the defects in the logs arises from basic knowledge of the structures of trees and defect in logs. The two-dimensional image can be described as a Z-θ plot of a log's properties and is created using a cylindrical projection of the known defects. In the image, each point corresponds to coordinates (Z,θ) that uniquely identify a ray, and the grayscale (or color) value at the point in the image depends on some scalar value evaluated for the corresponding ray.

For a conventional cylindrical coordinate system, the Z-axis is a straight line, and each value of coordinate Z defines an X-Y plane. The θ coordinate is an angle identifying a projection of a ray in the X-Y plane. In accordance with one embodiment of the invention, a modified cylindrical coordinate system has a Z axis that follows the center 125 of the growth rings 120 and accordingly bends as the position of growth center 125 changes from one value of Z to the next. For a further modification of the cylindrical coordinate system, each value of coordinate Z identifies a cone (e.g., cone 130) extending at an upward angle Φ. For angle Φ equal to zero, the cone is actually a plane as in a conventional cylindrical coordinate system. The upward angle Φ typically depends on the log or the species of tree that provided the log. For example, different trees or species of trees have limbs that grow at different characteristic angles with horizontal. For example, a pine tree may have limbs that grow nearly horizontal from the trunk of the tree, while hardwood trees have limbs with a distinct upward angle. Angle Φ can be selected according to the species of the log or according to evaluation of a particular log. With the

modified cylindrical coordinates, each value of coordinate Z identifies a plane or cone, and each value of coordinate θ identifies a ray in the plane or cone.

A two-dimensional data structure according to an embodiment of the invention and an image generated from the two-dimensional data array represents a set of values identifying properties of a log along rays corresponding to particular coordinate values Z and θ in the modified cylindrical coordinate system. A value in the data structure can be but is not limited to the average density over all or a portion of the associated ray, the severity of any defect on the ray, the distance from the center 125 or outside edge of log 100 to the defect, and similar measures of "good" wood, as opposed to defects. In accordance with another aspect of the invention, an embodiment of a data structure or image only includes features determined to be defects.

There are several ways to form images or data structures in accordance with the invention. One method determines the positions of defects using Cartesian coordinates Xc, Yc, and Zc and coverts those coordinates to the Z-θ space. Another method directly evaluates a desired property or scalar value along each ray corresponding to a Z-θ value.

Fig. 2 illustrates the relation between a log 200 and a data structure or image 205. In many cases the sawyer or log grader is not interested in the core of the log, and as is not interested in the structure of the bark on the log. Image 205 corresponds to an annular cylinder 250 having an inner limit R1 and an outer limit R2. Inner limit R1 is chosen to exclude less valuable core wood 255 from image 205. Outer limit R2 can be chosen to exclude bark and/or sapwood. Typically, outer limit R2 is at a set distance from the outside edge of the log so that the annular cylinder is not circular and has a varying thickness depending of the shape of the log. Alternatively, outer limit R2 can be a radius of fixed length. The property represented in image 205 is thus limited to a particular range of distances or percentages of the radius to ignore defects that are either too close to the center, or too close to the outside edge of the log. When the evaluated portions of the rays are limited, the Z-θ image 205 as illustrated in Fig. 2 can be roughly thought of as a tubular portion of log 200 that is unrolled and laid flat. This representation of tree structure is extremely efficient, reducing the volume of data from three dimensions to two and selecting the critical data. This simplifies manual or automated grading of the log and simplifies the optimization of a sawing or selecting a veneering pattern for log 200.

In general, analysis of the CT density data of log 200 indicates the presence of two types of defects, low-density and high-density. A low-density defect typically corresponds

to a crack 240 or a void area (not shown) in log 200. A high-density defect typically corresponds to a knot 210. The defects 210 and 240 appear in defect structure 205 and can be represented using different colors or grayscale levels, e.g., light shade or white for low density defects, a medium shade for no defects, and a dark shade for high density defects. A nearly unlimited number of variations for such images are possible based on the same underlying data structure.

The selection R1 and R2 limits image 205 to showing only the defects in the best wood 250. For example, crack 240 is predominantly in core wood 255 so that only a relatively small defect 245 appears in defect structure 205. A projection through the entire log onto a plane or a full radial projection would show a much larger defect area associated with crack 240. Further, radial projections of crack 240 from the geometric center of log 200 rather than from the growth center would spread the representation of crack 240 over a larger angular range since typical cracks pass through the growth center. An imbedded knot 230, which is in core wood 255 but does not extend to inner radius R1, would appear in a projection onto a plane or a complete radial projection. Image 205 does not include imbedded knot 230 because knot 230 does not extend into the region 250 of interest. Accordingly, the data structure and image 205 indicates the defects actually in the wood of interest in a manner that is precise and easy to interpret. A sawing strategy based on defect structure 205 can more easily provide a best sawing strategy for sawing the best wood 250 in log 200.

Fig. 3A is a flow diagram of a process 300 for generating a two-dimensional defect or data structure for a Z-θ image (e.g., image 205) from the CT data given in a three-dimensional Cartesian coordinate system Xc, Yc, and Zc. An initial step 305 identifies defects in the three-dimensional data structure using known processing techniques or the techniques described further below. The next step 310 selects the size or resolution of the Z-θ image. For example, if a pixel in the Z-θ image represents one centimeter in the Z direction and one degree of angle, the resulting image for a 4-meter log would then be 400 pixels high and 360 pixels wide (assuming the Z axis is vertical). A data structure accommodating the selected resolution of size is allocated for the image, and all points in the data structure are set to zero or a value representing a background color.

Step 315 selects one of the defects identified in step 305. Steps 320, 325, 330, 335, and 340 then calculate image data or pixel values for the selected defects. More particularly, step 320 determines the range of values of Cartesian coordinate Zc

corresponding to the selected defect (i.e., the Zc size or height of the defect) which will be converted into a range of values for the modified cylindrical coordinate Z. For process 300, the modified cylindrical coordinates employ rays originating from the grow center of the log. Alternatively, process 300 could use coordinates with a straight Z-axis or a Z-axis that follows the geometric center of the log or heartwood in the log.

Step 325 determines the position of the center of the annual growth rings at that Zc coordinates of the defect. Fig. 3B illustrates one method for finding the growth center. The method processes the density data for a cross-section 360 of the log to identify horizontal and vertical lines that cross the greatest number of annual rings. This method finds the approximate location of the geometric center 362 of cross-section 360 using conventional means and then selects a central area 366 around geometric center 362. In Fig. 3B, central area 365 is a square or rectangle. The growth center 364 is likely to be within central area 366 if central area 366 is appropriately sized for the species and diameter of the log.

Vertical lines 370 and horizontal lines 380 crossing central area 365 correspond to columns and rows of densities in the density array representing cross-section 360. The growth center can be identified from the sum of the absolute value of the gradient of the density or from the number of zero crossings of the density gradient. For each of lines 370 and 380, the sum of absolute values of the gradients (e.g., the sum of absolute differences in densities of adjacent voxels) depends on the number of times the density crosses the average density on that line. The sum of the absolute value or zero crossings of the density gradient can be determined for the entire length of the line or just for the portion of the line within central area 366.

Density plots 392 and 394 indicate the variation in density along lines 382 and 384 respectively. The horizontal line 384 with the highest sum of gradients or the most zero crossings in the density gradient crosses growth center 364, and indicates the Yc position of growth center 364. A more accurate determination of the Yc position of growth center 364 can be obtained by fitting the summed gradients for horizontal lines 380 to a curve and finding the peak of the curve. The process is repeated for vertical lines 370 summing absolute gradients (i.e., summing the absolute differences between densities of adjacent voxels in a column) or counting zero crossings in the density gradient to find the Xc position of growth center 364.

For some values of coordinate Zc, the determined Xc and Yc positions for the growth center 364 may be unreliable, especially if the corresponding slice of the log crosses several knots. For example, the determined Xc and Yc positions may be unreliable if there is no obvious peak in the x or y gradient data, or the detected center is at the edge of central area 366. The determined Xc and Yc positions may also be determined to be unreliable if the detected position differs significantly from the result of the neighboring values of coordinate Zc. If the determined Xc and Yc positions appear to be unreliable, the central area 366 can be expanded to widen the search for growth center 364. If the Xc and Yc positions are still unreliable, the reliable growth center coordinates for the various cross-sections of the log can be fit to a curve or combined in a local average (excluding or including the unreliable points). For most values of coordinate Zc, the calculation of Xc and Yc positions provides reliable results, and generally, the center of growth rings does not change quickly. Accordingly, if the determined Xc and Yc for one Zc value is determined to be unreliable (e.g., differ significantly from the average of the neighboring values of Zc), we can interpolate fit curve or use the local average to provide reliable growth center coordinate Xc and Yc.

Returning to Fig. 3A, when the surface corresponding to constant Z coordinate is a cone, the Z and Zc coordinates differ depending on the distance between the defect and the growth center. In particular, if Zc and R are the normal cylindrical coordinates with Zc-axis through the appropriate growth center, Z is less than Zc by R*tan(Φ) for an upward limb angle Φ as shown in Fig. 1. Step 330 calculates the range of coordinate Z for the selected defect assuming a constant position of the center of the growth rings, which is largely correct over the size of a typical defect such as a knot. Step 335 calculates the range of angle θ for the defect again using the identified center of the growth rings. Once the ranges for Z and θ are known, step 340 computes a scalar value for each (Z,θ) in the ranges. The scalar value ideally indicates the severity, depth, or type of the defect in a trajectory associated with the Z-θ coordinates. The trajectory can be a ray extending from the origin (e.g., the center of the growth rings) or may only extend between particular radii (e.g., R1 and R2 in Fig. 2). The determined scalar values are stored in the two-dimensional data structure associated with the Z-θ image.

Step 345 determines whether the last of the identified defect has been handled. If not, process 300 branches back to select another of the defects. After processing the last defect, defect areas have calculated data values and the non-defect areas have the

background values set during initialization of the data structure. Process 300 then continues to a segmentation step 350. Step 350 processes the data or defect structure to link contiguous defect pixels into defect groups and removes groups that do not contain enough pixels to be a significant defect.

Fig. 4 is a flow diagram of another process 400 for generating the Z-θ image or two-dimensional data structure in accordance with an embodiment of the invention. For each discrete Z position as selected in a step 410, a step 420 finds the center of the annual growth rings or the geometric center of the log or the heartwood as identified by a density threshold. At each Z position, step 430 selects a value θ identifying a ray or trajectory. The ray goes out at the specified angle θ as projected onto the X-Y plane and up at a limb angle Φ appropriate for the limbs of that log or the species of the tree that provided the log. The evaluated portion of the ray can extend from the center point or be limited to particular distances from the center point. Step 440 examines each voxel along the evaluated portion of the ray and computes a data value based on density, defect severity, distance from center, and previous data values. The determined data value is entered at the given Z,θ position in the image. Steps 450 and 460 respectively loop back to steps 420 and 410 so that the scalar values are determined for all θ and Z positions in the image.

Identification of defects using a threshold or other segmentation techniques based on the absolute magnitude of the density can be difficult because knots and other defects may have the same densities as some of the denser portions of clear wood. Sapwood in softwood species, for example, can be as dense or denser than knots. Also, knots in hardwood species often have a dense exterior with voids of decayed, low-density wood inside. Accordingly, the average density of a knot in hardwood may not indicate the presence of a defect. However, the knowledge that knots originate or extend outward from the center of the growth rings, and upward at an angle specific to the species, simplifies identification of defects. In particular, the ray in a direction characteristic of a knot will maximize the amount of high-density wood traversed and will have an average density higher than rays at angles not aligned with the knot. Similarly, rays that extend along radial cracks have lower average density than do other rays crossing through cracks. The representation in accordance with the invention thus makes defects standout more distinctly among the background of good wood.

As an alternative to picking average densities above or below a threshold to indicate defects, defects can be detected by identifying particular density patterns along a

ray. Using rays selected according to the structure of the log (e.g., from the growth center and at an upward angle characteristic of the log) makes the density patterns of defects more consistent and recognizable. For example, voids, or areas of decay inside the log, can be detected by following a ray inward. Starting from the outside of a log, a point along a ray is determined to be inside the log if there is a sufficient distance of wood of a selected density, between the point and the outside of the log. A region of very low density that is found after further movement toward the center indicates a defect.

Observation of variations in the average densities along adjacent rays can also improve ability to distinguish defects from clear wood. In particular, variations in the width of the sapwood region of a log can make the average density for a ray through a thick portion of the sapwood similar to the average density for a ray passing through a small knot and a thin section of sapwood. Accordingly, in this situation, having a fixed threshold average density as the sole indicator of a defect would either miss the small knot or incorrectly identify the region with thick sapwood as a defect region. In accordance with an aspect of the invention, defects can be identified by subtracting a local average from each point in the Z-θ image and then identifying the defects for rays having an average density that differs from a local average density by more that a threshold amount. This provides better discrimination of defects because the thickness of the sapwood changes slowly with both angle θ and distance Z. Accordingly, an area in the Z-θ image having an average density that differs from the average density in a surrounding area of the Z-θ image indicates a defect.

Once a ray is identified as a defect ray, the depth of the defect can be determined approximately by following the ray inward, moving past the bark and sapwood, until high densities or voids are encountered. Fig. 5 is a flow diagram illustrating a process for tracing a ray and identifying defects along the ray. An initial step 510 selects a trajectory for ray tracing. The trajectory corresponds to constant Z and θ and terminates at the growth center of the log. A limb angle Φ is appropriate to the species being examined. This trajectory has Cartesian coordinates that are a function of the distance from the growth center.

Step 515 determines a step vector having a length S and Cartesian components Sx, Sy, and Sz according to Equations 1.

Equations 1: $Sx = S*\cos(\Phi)*\cos(\theta)$.

$Sy = S*\cos(\Phi)*\sin(\theta)$.

$$Sz = S*\sin(\Phi)$$

For each step along the ray, the step vector will be added or subtracted from the Cartesian coordinates a point on the ray to determine the next point for consideration on the ray. An alternative to using fixed limb angle Φ for Sz is a lookup table for limb trajectory. Using a lookup table allows for a curve in the upward growth of the limb (e.g., a changing limb angle). Step 525 selects a starting point that is a distance R along the ray. The distance R can be a fixed value, but in the illustrated process 500, step 520 selects as the distance R the maximum distance from growth center to outside edge of the log at angle θ. Step 530 initializes the Cartesian coordinates of the current point to the point the distance R along the selected ray. Starting at this initial point, process 500 moves inward along the ray in steps of having components Sx, Sy, and Sz, until a specified distance from the growth center is reached.

For each step point along the ray, step 530 finds Cartesian coordinates (Xc,Yc,Zc) of the point and finds the corresponding density D for that point from the three-dimensional data structure representing the log. The use of the density depends on whether process 500 has determined that the current point is "inside" the log. Decision step 535 determines whether the current point is inside the log. The initial or starting point is not inside the log so that process 500 initially branches from step 535 to step 540. Step 540 determines whether the density D is greater than a threshold density D0 characteristic of wood inside a log of the type being analyzed. If in step 540, density D is less than threshold density D0, the current point is outside the log, and process 500 branches from step 540 to step 560 for the next step along the ray. If in step 540, density D is greater threshold density D0, step 545 increments a count of pixels having a density greater than threshold density D0. Then, if in step 550, the count is greater than a threshold count C0, step 555 sets a flag to indicate that process 500 has reached a point inside the log. The threshold count C0 typically depends on the characteristics of the log (e.g., the typical thickness of bark) and the length S of each step.

If the "inside" flag is set, step 535 determines the current pixel is inside the log, and process 500 branches from step 535 to decision step 570, which determines whether density D is less than a low threshold density DL. The low threshold density DL is typically fixed according to the type of wood, but alternative threshold density DL can be selected according to the average densities in surrounding areas of the log. A point that is inside the log and of a low density indicates a void or decay defect, and step 575 marks

such points as defects. More particularly, in the two-dimensional defect structure, the data corresponding to the Z and θ coordinates of the current ray is set to indicate a void type defect and can further include additional information such as the distance along the ray to the defect. After marking the defect in the two-dimensional data structure, tracing process 500 can terminate for the ray.

If density D is not less that the low threshold DL, step 580 determines whether the density is greater than a high threshold density DH. The high threshold density DH is typically fixed according to the type of wood, but alternative threshold density DH can be selected according to the average densities in surrounding areas of the log. If the density D is higher than the high threshold density DH, step 585 updates the two-dimensional data structure to indicate a high-density defect for the pixel corresponding to the ray being traced. If density D is neither higher than DH nor lower than DL, step 590 adds density D to an accumulator.

After step 590, step 560 checks whether the current step is the last step, e.g., tracing has reached a target distance on the ray. If tracing of the ray is not complete, step 565 changes the current coordinates by the amount of the step vector before step 530 looks up a new current density D. If tracing is complete, step 595 divides the accumulator by the number of points detected to form an average density for the ray and sets the appropriate data value in the data structure. The average density can be used in the manners described below to determine whether a defect is present on the ray even when no single voxel on the ray has a density below DL or above DH.

The tracing process 500 can be repeated for each value of Z and θ in the desired range. The resulting two-dimensional defect structure can be further processed to detect defects that may be hidden by density variations that occur for example, in a log having sapwood that varies in thickness. Fig. 6 illustrates such a process for identifying hidden defects. In process 600, an initial step 610 copies the data structure including average densities for each ray not previously identified as crossing a defect. Step 620 selects values Z and θ that do not correspond to a previously identified defect, and step 630 calculates an average density from the density values in area (e.g. within 10 degrees and 10cm) of the selected Z and θ in the copy. Step 640 determines whether the density for the selected Z and θ significantly differs from the average density (e.g., a percentage difference greater than about 10% or an absolute difference of about 0.1 or 100 CT counts or Hounsfield units). A significant difference indicates a defect, and step 650 sets the data

value corresponding to the selected Z and θ in the original data structure to indicate the defect. If there is no significant difference, step 660 sets the data value corresponding to the selected Z and θ in the original defect structure to zero or to otherwise indicate no defect. The resulting defect structure has zero values everywhere except at coordinates corresponding to defects. Such data structures, which are mostly zeros, can be easily compressed for transmission or storage.

Optionally, the defect structures can be further processed. Segmentation for example, can collect groups of adjacent defects into a single defect having a size and shape depending on the defect points collected. Segmentation can also discard isolated defects that are too small to be a problem in the desired application of the log. Further processing can also retrace the rays identified as being defects to find an approximate distance from the center to each defect.

A computer or human can use the Z-θ defect structure for grading a log. By reducing the data describing the log from three-dimensional to two-dimensional, the data volume is vastly decreased, and the data is easily viewable as an image. Many log grading systems are currently in use or may be developed, but in general, the distribution of defects is of key importance to grading and is typically more important than the total number of defects. The distribution of defects is important because the eventual grade and value of a board is based on the length and width of areas of clear lumber that can be extracted from the board by cutting away the areas containing knots or other defects. A log having several defects aligned along one or two angles θ can be aligned and cut to provide high value boards that avoid the defects. Accordingly, such logs are of much higher value than a log having the same number of defects at randomly distributed locations. Similarly, if the defects are concentrated at the ends of the log, the log is of higher value because sufficiently long sections of clear wood can found in the middle of the log.

Visual inspection of the Z-θ defect map quickly indicates the distribution of knots both along the length of the log (Z direction) and angular position θ. An inspector with some training can quickly determine a log's value from the length and width of defect-free sections in the Z-θ defect map. The alternatives of looking at each CT slice in succession or rotating a three-dimensional rendering are slow and require complex reconstructions and spatial awareness.

An automated grading process can similarly be based on cylindrical projections such as the Z-θ defect map to produce an objective, quantitative valuation of a log based

on a flexible and adaptable set of rules. The rules are modified depending on the species and intended purpose of the log. In particular, a sawyer seeking boards having particular mix of widths, thicknesses, and or lengths can adopt a set of grading rules that grade a log according to the value of boards of the desired mix sawn from the log. The automated grading determines how many sections of the log can produce a defect-free board at least W units wide and L units long. The Z-θ defect map indicates the length and width of clear sections. The angular distance between defects on either side of a clear section in Z-θ defect map and the depth of the wood evaluated (e.g., R2-R1 in Fig. 2) indicate the potential thickness of a clear board. If a cut is a distance Dc from the center of the log and perpendicular to a radius of the log and the A is the angular distance between the defects, the width of clear wood in the board is about 2*Dc*tan(A/2). The clear length of the board is the Z distance between defects in the Z-θ defect map.

Figs. 7A and 7B illustrate a grading process 700 of a log based on the number and distribution of defects in a Z-θ defect map 750. In the exemplary process 700 illustrated in Figs. 7A and 7B, the grading rules select a single width W, and two lengths L1 and L2 for clear sections of board. L2 is longer than L1. Additionally, a clear section having an area greater than or equal to W*L2 is assigned a value V2, and a smaller clear section having an area greater than or equal to W*L1 is assigned a value V1. Alternatively, the assigned value can be a function of the actual length and width of a clear section.

Grading process 700 in an initial step 705 selects an angular distance Ar that produces a board of width W at a cut distance Dc from the center of the log. Process 700 in step 710 searches through Z-θ defect map 750 for a defect-free angle group. A defect-free angle group is a group of A data values that are consecutive in the θ direction (e.g., in a row of the two-dimensional data structure) and indicate no defect. For each defect free angle group found, step 715 of process 700 scans Z-θ defect map 700 in the Z direction (e.g., along columns in the two-dimensional data structure) and finds the length L of a section that is free of defects in all A columns. Alternatively, if the Z-θ defect map 750 contains depth information for defects, the process finds the length of each section having no defects that are greater than the cut distance Dc from the center of the log. If step 720 determines the section associated with an angle group is length L2 or longer, step 725 adds a score V2 to a grade value. Otherwise, if the length L is shorter than L2 and step 730 determines the length L at least as long as length L1, step 735 adds a score V1 to a grade value. Step 740, which follows steps 725 and 735, removes data values in the section from

further searching for angle groups so that each defect-free angle group is in at most one section that contributes to the grade value. After completing the searches for defect free angle groups, the total grade value indicates a grade for the log.

The preceding grading process produces an approximate value for a log with no recommended sawing strategy. Efficient sawing of the log might produce a corresponding value for the resulting boards, but a poor sawing job might produce less value. A grading process that accounts for a sawing strategy is inherently more accurate. In particular, a grading process combined with a process to optimize the sawing strategy produces a grade value estimate that more accurately indicates the value of the log.

The sawing strategy can indicate the orientation of a log for each pass through a saw. Frequently, the only freedom provided in a sawing strategy is the orientation of the log during the first cut. Even when a sawing strategy permits additional freedom in selection of cutting parameters, the orientation for the first cut from a log often has the largest affect on the value of the boards produced. Accordingly, the value produced for an optimum initial orientation indicates an optimized grade for a log. In accordance with another aspect of the invention, the Z-θ defect map facilitates optimization of the sawing strategy.

Optimization of the sawing strategy can be done manually based on viewing the image of the Z-θ defect map. In particular, the Z-θ defect image shows the location of the log defects in both Z and θ directions. A person viewing a single Z-θ defect image can select the orientation angle that provides the largest zone of clear wood, or the orientation in which there is the largest area of clear zones. More specifically the first cut can be selected to be perpendicular to the log radius directed at an orientation angle θo that is in the center of the largest clear section in the Z-θ defect image.

Graphical tools applied to a computerized Z-θ defect image enhance the ability of a person to visualize the range of angles that constitute the boards produced by a sawing strategy. For example, Fig. 8A illustrates an example of a sawing strategy for a log 800. Accordingly to this sawing strategy, a cut 810 at an orientation angle θo opens a first face FACE1 in log 300. A cut 820 at orientation angle θo+180° opens a second face FACE2. A cut 830 at orientation angle θo+90° opens a third face FACE3, and a cut 840 at orientation angle θo+270° opens a fourth face FACE4. Many other sawing strategies are possible, but the strategy of Fig. 8A provides an example for illustration of an aspect of the invention.

For each face FACE1 to FACE4, each board correspond to a range of angles that depends on the cutting distance Dc and the diameter of log 800. The angular range changes if the cutting distance Dc changes, for example, for subsequent cuts at the same orientation. Fig. 8B illustrates a graphical tool applied to a Z-θ defect image 850 to highlight angular ranges corresponding to boards at each face. The graphics tool adds boundary lines 851 to 858 marking the angular ranges corresponding to each face. Faces FACE1, FACE2, FACE3, and FACE4 correspond to the angular ranges between respective pairs of boundary lines 851 and 852, 853 and 854, 855 and 856, and 857 and 858. A grader or sawyer can shift boundary lines 851 to 858 relative to Z-θ defect image 850 to shift as many defects as possible into unused angular ranges (e.g., between boundary lines 858 and 851, 852 and 853, 854 and 855, or 856 and 857) and provide the largest clear section in the area corresponding to the first face FACE1. Once the boundary lines are shifted to the desired locations, the angle at the center of FACE1 indicates an optimal cutting angle θo.

The process of optimizing cutting angles based on a cylindrical representation of defects can be automated. The Z-θ defect structure facilitates efficient optimization of sawing orientations by vastly reducing the volume of data processed. Fig. 9 illustrates a grading and optimization process 900 that accounts for the sawing strategy used. An initial step 905 of grading process 900 determines the angular range Ar corresponding to a desired width of clear board. A step 910 then selects the sawing strategy that will be employed on the log, and step 915 selects an initial orientation for the log when the cutting begins. The sawing strategy and the orientation of the log together define the faces that result from cutting the log. Step 920 selects from the defect structure a data block corresponding to a face encounter during the cutting. More specifically, the data block corresponds to an angular range defined by the face selected in step 920. If the defect structure includes depth information, the data block represents defects at the depth corresponding to the face. Step 930 then determines a grade value for the face using, for example, the grading process described above in regard to Figs. 7A and 7B (but limited to the block corresponding to the selected face.

A loop including selecting a data block for a face (step 920) and determining a grade for the face (step 935) can be repeated for each face defined by the sawing strategy or until step 935 otherwise determines that no further faces need to be considered. Generally, when the defect structure does not contain depth information, not all faces need

to be considered. For example, the number of face evaluated can be equal to the number of times that the sawing strategy changes the orientation of the log (e.g., four faces for the cutting strategy of Fig. 8A). Step 940 combines the grade values for the faces evaluated to determine a grade value corresponding to the orientation selected in step 915. In the exemplary embodiment, step 940 sums or averages the grade values for the faces considered.

A loop including steps 915, 920, 935, and 940 repeatedly determines grade values for different orientations of the log until step 945 determines no further orientations need to be considered. Step 950 sets the grade for the log equal to the best grade achieved for any of the orientations. The orientation achieving the highest grade is the recommended initial cut orientation.

Fig. 10 is a block diagram of a system 1000, which can implement many of the above-described process. System 1000 includes a CT scanner 1010, a computer 1020 with a display 1030, and a computer 1040 with a display 1050. CT scanner 1010 scans a log to produce raw data which conventional CT techniques convert a series of two-dimensional data structures or density distributions corresponding to a series of slices or cross-sections of the log. The collection of all of the two-dimensional data structures together form a three-dimensional data structure 1027. Three-dimensional data structure 1027 can be stored in a memory 1024 of computer 1020 for processing by a processor 1022 executing analysis software 1026. Analysis software 1026 can implement the any of above described automated processes including converting three-dimensional data structure 1027 to a two-dimensional data structure (or defect structure) 1028. A user can view the two-dimensional data structure as an image on display 1030.

Although computer 1020 has access to three-dimensional data structure, such access is not require for analysis of the properties of a log. In particular, computer 1040 has analysis software 1046 that processor 142 executes analysis software 1046 to process a two-dimensional data structure 1048 in memory 1044. In particular, analysis software 1046 can process data structure 1048 for viewing on display 1050, log grading, or optimizing sawing of a log. Since two dimensional data structure 1048 is much more compact (and is easily compressed) transmission of the two dimensional data structure 1048 to computer 1040 typically takes fewer resources and less time than would transmission of a three dimensional data structure. Accordingly, a user of computer 1040, for example, a log buyer at a sawmill can evaluate a log for a particular sawing strategy or

using particular grading criteria without ever needing to deal with the amount of data in the three-dimensional data structure. Different sawmills with different sawing strategies or different valuations of boards can easily and efficiently evaluate logs based on the sawmills' specific needs.

Below is the source code for various computer programs, written in C++, that may be used in conjunction with the methods and structures described herein.

```
/*
This file contains C++ code for the following computational methods:

  BuildRadialView:     Create selected radial image and copy to display
image
  RadialProjection:    Create Z-Theta projection of image data
  SegmentRadialImage: Perform local thresholding of radial image to find
defect
                       pixels; Group contiguous defect pixels into defect
groups
  GradeRadialImage     Analyze radial image to find grade of log and
recommended
                       orientation angle for first cut
  SegmentImage         Generic image method for segmenting an image to
groups
                                of contiguous pixels with a value
greater than some threshold.
  AnalyzeDefects       Analyze defect groups to find depth information
  FollowPith               Find centers of annual growth rings,  Remove
points of
                                no confidence, and smooth the centers to
find the pith
                                of the tree
  FindPith                Find the center of the annual rings for a
single CT image
  DisplayCurrentcut    Display method for showing cut positions in a
radial image
*/


////////////////////////////////////////////////////////////////////////////
////
// Create selected radial view and copy into display image
void vTree::BuildRadialView(RadialImg* radImg, int radialType ){
  int iz,iy;
  double BScale=(double)360.0/radImg->Height();
  double ZScale=(double)radImg->Width()/(double)(NSlices());


  radImg->BScale(BScale);
  radImg->ZScale(ZScale);
  radImg->SetUnits(mmPerSlice,mmPerPix);
  radImg->NSlices(NSlices());
  radialImgType = radialType;


  // create Radial Projection
```

```
    if (!radialP) {
          RadialProjection();

    }
    BDINT *dataP=radialP->data;

    // perform local threshholding to distinguish defects from background
    // Segment thresholded image to find each defect
    if ( radialImgType ==2 || radialImgType==3) {
          SegmentRadialProjection();
          dataP=radialThreshP->data;
    }

    // Analyze each defect to find depth size, and type of defect
    AnalyzeDefectList();

    // analyze segmented radial image for log grade and optimum
orientation
    if (radialImgType==3) {
          GradeRadialProjection();
          dataP=radialThreshP->data;
    }


     // resize and copy radial image into display image
     for (iz=0; iz<radImg->Width(); iz++) {
           int tz=iz/ZScale;
           double yfac=radialP->Height()/(double)radImg->Height();
           if (tz<NSlices()) {
                 for (iy=0; iy<radImg->Height(); iy++){
                       int ty=yfac*iy;
                       radImg->data[iy*radImg-
>Width()+iz]=dataP[ty*NSlices()+tz];
                 }

           } else {
                 // zero out remaining columns
                 for (iy=0; iy<radImg->Height(); iy++)
                       radImg->data[iy*radImg->Width()+iz]=0;
           }
     }
     sendWVStatusMessage( statusWind, "Finished Radial View",
WV_STATUSBAR_SECTION_1);
     return;
}

////////////////////////////////////////////////////////////////////////
////
// Create polar projection for each CT slice in the tree data structure.
// by tracing rays originating from the pith, if detected, or the
geometric
// center of the slice.
void vTree::RadialProjection(){
      if (radialP) return;

      // Find the center of the annual rings for each slice
      FollowPith();

      // create status bar showing progress.
      createWVProgressWindow("Calculating Radial View:", 0, NSlices(),
1);
```

```
        // allocated image for projection
        if (!radialP) radialP=new RadialImg(NSlices(),360);
        BDINT *dataP=radialP->data;

        int islice;        // CT slice index
        int iangle;        // angle index
        int ix,iy,iz;      // x,y, and z indexes used for ray tracing

        // radii is the average radius of the log over all slices at each
of 360 angles
        pu_bzero((WVPOINTER)radii,sizeof(radii));
        int lastRadius[360];   // radius of previous slice at each angle

        double rayTraceStep=1;  // step size, in pixels, along ray when
tracing

        // outerThicknessCM: e.g. 2cm
        // outerThickness is the region near the outside edge of the log
that will be
        // ignored (bark, for example).  outerThickness is converted to
number of steps
        // along ray
        int outerThickness=ConfigP-
>data().OuterThicknessCM*10/(MmPerPix()*rayTraceStep);


        // create polar projection for each slice (in Z direction)
        for (islice=0; islice<NSlices(); islice++) {
            CtImg *sliceP=&imgPs[islice];
            // indicate progress
            updateWVProgressWindow();

            // for each angle, trace a ray
            for (iangle=0; iangle<360; iangle++) {
                int sum=0;                    // accumulator for sum
of values along ray
                int averageCT=0;              // average CT value alon
the ray
                int maxVal=0;                 // Maximum value along a
ray
                int depth=0;                  // distance, in steps,
from edge of log when
                //                                    ray tracing
starts
                int rayLength=0;              // length of traced ray,
in pixels

                //rx, ry, rz: accumulators for x,y,z coordinates along
ray as it
                // is traced.  rx, ry, rz initialized to ray origin,
which is detected
                // pith (or geometric center if not detected)
                double rx=sliceP->Pithx();
                double ry=sliceP->Pithy();
                double rz=islice;

                double rAng=iangle*D2R; // current angle in radians
                // dx,dy,dz: x,y,and z distance, in voxels, between
each point along ray
                double dx=cos(rAng)*rayTraceStep;
                double dy=-sin(rAng)*rayTraceStep;
```

```
                    // limb angle deg: e.g. 2 deg for spruce, 50 deg for
alder
                    double limbAngleRad=limbAngleDeg*D2R;
                    double dz=tan(limbAngleRad)*rayTraceStep;
                    dz*=mmPerPix/mmPerSlice;   // adjust for voxel
dimensions

                    // depth of penetration (in steps) of creosote, or
other material,
                    // used for evaluation of telephone poles.
                    int penetrationDepth=0;

                    // maxRadius: maximum length of ray, in rayTraceSteps,
from
                    // origin of the ray.  maxRadius is constrained by
                    // image size, and z position (if limb angle is
nonzero)
                    int maxRadius;
                    int minRadius;    //stop tracing when inner radius is
reached.
                    int ir;          // current number of steps from origin

                    minRadius=ConfigP->data().InnerRadiusCM*10/MmPerPix();
// e.g. 5cm

                    //start from outside of image, and move toward the
center.
                    //after first slice, save time by using outside edge
of previous slice
                    //
                    if (islice==0) lastRadius[iangle]=Width();
                    maxRadius=lastRadius[iangle]+10;

                    // limit the maximum number of steps by the image size
                    if (dx>0) maxRadius=MIN(maxRadius,(Width()-rx)/dx);
                    if (dx<0) maxRadius=MIN(maxRadius,-rx/dx);
                    if (dy>0) maxRadius=MIN(maxRadius,(Height()-ry)/dy);
                    if (dy<0) maxRadius=MIN(maxRadius,-ry/dy);
                    if (dz>0) maxRadius=MIN(maxRadius,(NSlices()-
islice)/dz);
                    if (dz<0) maxRadius=MIN(maxRadius,(islice+1)/dz);

                    // start from maximum ray length and trace toward the
center of the
                    // log.  Detect outside edge of log by thresholding
(pixels outside
                    // of the log have already been removed).  Travel into
the log along
                    // the ray until <outerThickness> pixels are above a
threshold.
                    ir=maxRadius;
                    rx+=ir*dx;
                    ry+=ir*dy;
                    rz+=ir*dz;

                    for (; ir>minRadius; ir--) {
                        ix=rx;
                        iy=ry;
                        iz=rz;
                        rx-=dx;
                        ry-=dy;
```

```
                        rz-=dz;
                        //sliceP=&imgPs[iz];

                        WVUINT16 val=(imgPs[iz])[iy*Width()+ix];
                        if (val>150) {
                              depth++;
                              sum+=val;

                              if (val>ConfigP->data().TreatmentCT)
penetrationDepth++;

                              // are we far enough into the log to start
                              // accumulating?
                              if (depth>outerThickness) break;
                        }//if
                  }//for

                  lastRadius[iangle]=ir+outerThickness;
                  radii[iangle]+=ir+outerThickness;

                  // Two quantities determine how long to trace the ray.
minRadius is
                  // already set to the inner radius, which represents
the core of the
                  // tree that may not be of interest.  MaxThickness is
the maximum
                  // length of the ray, starting from where we are now.
                  minRadius=MAX(minRadius,ir-ConfigP-
>data().MaxThicknessCM*10/MmPerPix());

                  // Countinue in, looking for voids this time
                  sum=0;
                  rayLength=ir-minRadius;
                  for (; ir>minRadius;ir--) {
                        ix=rx;
                        iy=ry;
                        iz=rz;
                        rx-=dx;
                        ry-=dy;
                        rz-=dz;
                        depth++;
                        //    sliceP=&imgPs[iz];
                        WVUINT16 val=(imgPs[iz])[iy*Width()+ix];

                        //make voids look dense, so we see both dense
and void areas
                        // as defects.
                        if (val<ConfigP->data().RadialVoidCT)
                              val=2000;

                        // Another option: look for maximum value along
the ray.
                        if (ConfigP->data().RadialShowMax) {

                              if (val>maxVal) maxVal=val;
                        } else {
                              if (val>ConfigP->data().TreatmentCT)
penetrationDepth++;
                              sum += val;
                        }

                  }//for
```

```
                    // compute average CT value along the ray.
                    if (rayLength>0) averageCT=sum/rayLength;

                    // select one of several options for the value placed
in the projection.
                    // Maximum value of the ray
                    if (ConfigP->data().RadialShowMax)
                        dataP[islice+NSlices()*iangle]=maxVal;
                    // average value if the average value was greater than
a threshold.
                    else if (ConfigP->data().RadialMinAvg)

        dataP[islice+NSlices()*iangle]=averageCT>ConfigP-
>data().RadialMinAvg?averageCT:0;
                    // For telephone poles, the depth of preservative
penetration
                    else if (ConfigP->data().MinPenetrationMM) {
                        double minPentrationPix=ConfigP-
>data().MinPenetrationMM/MmPerPix();
                        int pval=500+penetrationDepth*30;
                        if (penetrationDepth<minPentrationPix) pval=0;
                        dataP[islice+NSlices()*iangle]=pval;
                    }//else if
                    else {
                        // normal case: average value along the ray.
                        dataP[islice+NSlices()*iangle]=averageCT;
                    }
                }//for on angles

    }//for on slices

    // Save projection to a file
    if (ConfigP->data().SaveRadialImage) {
        int imgBytes=360*NSlices()*sizeof(BDINT);  //BDINT is 16 bit
integer
        puFileSave("radial1",(WVPOINTER)dataP,imgBytes);
    }

    // Compute average radius for the log at each angle.
    for (iangle=0; iangle<360; iangle++) radii[iangle]/=NSlices();


    //clean up progress window
    endWVProgressWindow();
}

// This method performs local thresholding of the radial image,
// and segments the image to group adjacent defect pixels into defects
void vTree::SegmentRadialProjection() {
    int iangle,slice;

    // delete previous results and allocate new image
    if (radialThreshP) delete radialThreshP;
    radialThreshP=new RadialImg(NSlices(),360);

    // pointers to the image data of projection data and (new)
threshold data
    BDINT *projDataP=radialP->data;
    BDINT *threshDataP=radialThreshP->data;

    int startAngle;
```

```
        int ndefects;

        // establish size of rectangle used for local averaging
        int boxSizeA=ConfigP->data().RadialAvgBoxSize;
        int boxSizeZ=boxSizeA;      //I guess it's square for now


        //divide the image up into stripes containing <boxSizeA> angles.
        // perform local averaging along each stripe.  This is
significantly
        // faster than treating each point individually.
        for (startAngle=0; startAngle<360; startAngle+=boxSizeA) {
              int stopAngle=startAngle+boxSizeA-1;
              if (stopAngle>=360) {
                    stopAngle=359;
                    boxSizeA=stopAngle-startAngle+1;
              }
              int nBoxPoints=boxSizeA*boxSizeZ;


              // find sum and average of all points in the box at the
beginning of
              // the stripe.
              int endslice;
              int sum=0,avg;
              int startSlice=0;
              for (iangle=startAngle; iangle<=stopAngle; iangle++) {
                    for (endslice=startSlice; endslice<boxSizeZ;
endslice++) {
                          sum+=projDataP[endslice+iangle*NSlices()];
                    }//for
              }//for
              avg=sum/nBoxPoints;

              // slide the box along in z, one slice at a time,
              // and recompute average value in the box.
              for (slice=0 ; slice<NSlices(); slice++) {
                    // leave box in place for first and last <boxSizeZ/2>
points.
                    // otherwise, subtract the beginning points, and add
the new
                    // ending points to the sum.
                    if ((slice>=boxSizeZ/2) && (slice<NSlices()-
boxSizeZ/2)) {
                          for (iangle=startAngle;
iangle<startAngle+boxSizeA; iangle++) {
                                sum-=projDataP[startSlice +
NSlices()*iangle];
                                sum+=projDataP[endslice +
NSlices()*iangle];
                          }//for
                          endslice++;
                          startSlice++;
                          avg=sum/nBoxPoints;
                    }//if

                    // Now subtract average of box from each point in the
box of
                    // a vertical line at the center of the box.  If this
value is
```

```
                    // greater than a threshold, make the point in the
thresholded image
                    // non-zero; otherwise set the threshhold image point
to zero.
                    for (iangle=startAngle; iangle<startAngle+boxSizeA;
iangle++) {
                        int idx=slice+NSlices()*iangle;
                        int val=projDataP[idx];
                        if ((val-avg)<ConfigP->data().RadialMinDiff)
                            threshDataP[idx]=0;
                        else
                            threshDataP[idx]=1800;
                    }//for
                }//for
        }//for
        if (ConfigP->data().SaveRadialImage) {
            int imgBytes=360*NSlices()*sizeof(WVUINT16);
            puFileSave("radial2",(WVPOINTER)threshDataP,imgBytes);
        }

        int minDefectSize=ConfigP->data().RadMinDefectPix;   //e.g. 10
pixels
        if (minDefectSize) {
            // Find groups of adjacent defect pixels, and group them
into
            // defects.  Remove those defects with fewer than
<minDefectSize>
            // pixels in the group,  Add each defect to defect list
            ndefects=radialThreshP-
>SegmentImage(0,minDefectSize,&DefectsList);
        }

        if (ConfigP->data().SaveRadialImage) {
            int imgBytes=360*NSlices()*sizeof(BDINT);
            puFileSave("radial3",(WVPOINTER)threshDataP,imgBytes);
        }


}//RadialProjection

void vTree::GradeRadialProjection() {
    int slice,startAng,ix;
    int ACount=0,BCount=0,BTotal=0;
    BDINT *dataP=radialThreshP->data;
    int angDel=ConfigP->data().GraderAngleInc;   // e.g. 3 deg
    int scoreCount=1;
   WVREAL32 segmentScores[360];
 pu_bzero((WVPOINTER)segmentScores,sizeof(segmentScores));
     int scoreIx=0;
    float logScore=0;

      //remove previous colors from segmented image, leaving only
      //pixels non-zero.
    for (slice=0; slice<NSlices(); slice++) {
            for (startAng=0; startAng<360; startAng++) {
                if (dataP[startAng*NSlices()+slice]<=700)
                    dataP[startAng*NSlices()+slice]=0;
            }
    }
```

```
        // For 360 degrees of rotation, select segments <angDel> degrees
    in width.
      for (startAng=0; startAng<360; startAng +=angDel,scoreIx++) {
                int tBCount=0;
                int colorVal;
                int Acolor=700;
                int Bcolor=400;
                int AScore=0;
                int BScore=0;
                int cuttingStart=0;
                float cuttingLengthCM=0;
                float CmPerSlice=MmPerSlice()/10;

                int testAng;
                // for each segment, look along the length of the segment to
    find
                // all of the clear cuttings in which there are no defect
    pixels in the segment.
                int stopAng=startAng+angDel;
                if (stopAng>360) stopAng=360;
                for (slice=0; slice<NSlices(); slice++) {

                      int isClear=TRUE;  // assume segment is initially
    clear

                      //cutting must end at the end of the log.
                      if (slice==NSlices()-1) isClear=FALSE;
                      else {
                            for (testAng=startAng; testAng<stopAng;
    testAng++){
                                  //Not at the end, so look through all
    angles of the segment
                                  //for a defect at the current slice
                                  if (dataP[slice + testAng*NSlices()]) {
                                        isClear=FALSE;
                                        // defect found, no need to look at
    the rest of the angles
                                        break;
                                  }
                            }
                      }


                      if (isClear) {
                            //no defect found, so increment the length
                            cuttingLengthCM+=CmPerSlice;
                      } else {
                            // A defect was found, so the cutting is no
    longer clear.  How long
                            // was the clear part?  Here we choose two
    lengths - lengthA and
                            // lengthB.  LengthA>LengthB, and each length
    has a score.  If the
                            // cutting is longer than LengthA it gets a A
    score calculation. If
                            // between A and B length it gets B score
    calculation.  if it is less
                            // than LengthB it gets no score.
                            // scores are a simple linear polynomial:
                            //   C1 + C2*(Length-minLength)
                            // examples:
                            // ALengthCM: 200  AVal1: 300  AVal2: 1.5
```

```
                        // BLengthCM:  75  BVal1:  75  BVal2: 0.9
                        if (cuttingLengthCM>=ConfigP->data().BLengthCM)
    {
                            if (cuttingLengthCM>=ConfigP-
    >data().ALengthCM) {
                                ACount++;
                                AScore+=ConfigP->data().AVal1+
                                    ConfigP-
    >data().AVal2*(cuttingLengthCM-ConfigP->data().ALengthCM);
                                colorVal=Acolor;
                            } else {
                                BScore+=ConfigP->data().BVal1+
                                    ConfigP-
    >data().BVal2*(cuttingLengthCM-ConfigP->data().BLengthCM);
                                colorVal=Bcolor;
                                tBCount++;
                            }

                            // Apply coloration to the image to
    indicate which sections are
                            // A or B length.
                            while (cuttingStart<slice) {
                                for (testAng=startAng;
    testAng<stopAng; testAng++){
                                    dataP[cuttingStart +
    testAng*NSlices()]=colorVal;
                                }
                                cuttingStart++;

                            }
                        }
                        cuttingLengthCM=0;
                        cuttingStart=slice+1;
                    }
                }

                // At the end of the segment, calculate score for the entire
    segment.  This
                // is sum of the scores for all of the segments.
                if (tBCount) BCount++;
                BTotal+=tBCount;

                // record the score for this segment.
                segmentScores[scoreIx]=AScore+BScore;
                scoreCount=scoreIx+1;

                //accumulate the score for the entire log.
                logScore+=AScore+BScore;
    }

    // Done with all of the segments.  Compute various metrics for grade
    of the log
    // that are not based on a cutting strategy.
    logScore/=scoreCount;
    double aa=ACount;
    double bb=BCount;
    double cc=5*aa+bb;
    double dd=100*cc/(5*360/angDel);

    // Compute a log grade based on the defects found using the radial
    image,
```

```
    // not the radial image itself.
    char GradeCode;
    int sumV;

    LOG_GRADE grade=GradeLog(&sumV);
    if (grade==GRADE_A) GradeCode='A';
    else if  (grade==GRADE_B) GradeCode='B';
    else GradeCode='C';
    printf("S=%5.1f (%5.1f) A=%3i B=%3i totalB=%3i\n",
          logScore,dd,ACount,BCount,BTotal);
    FILE *fd=pu_fopen("$(DATADIR)/temp/grader","a");
    if (fd) {
          printf("aa=%f bb=%f cc=%f dd=%f (%f) ndef=%i\n",aa,bb,cc,dd,
                ((float)ACount*5+(float)BCount)/(float)(360/angDel*5),
                NbDefectsIdentified());
          fprintf(fd,"S=%5.1f (%5.1f) A=%3i B=%3i totalB=%3i ndef=%i g=%3i
(%c) %s\n",
                logScore,
                dd,ACount,BCount,BTotal,
                NbDefectsIdentified(),
                sumV,GradeCode,
                Id());
          fclose(fd);
    }

    // find scores for the opening face of each rotation angle of the log.
An opening
    // face represents an angular range (GraderFaceAngle) that includes
some number
    // of cutting segments.   Subsequent faces will generally represent a
smaller
    // angular range.  Here we compute the value for each face size at
each angle of
    // log rotation.
    // We have already computed the score for each cutting, so we just add
up the values
    // of the cuttings that are contained in the face angle.  The size of
the face
    // angle is determined by the cutting strategy selected.
    float maxScore=0;
    float maxF1Score=0;
    int maxIx=0;
    int maxF1Ix=0;
    int lix;
    double bestAngle=0;
    int f1chunks=0.5*ConfigP->data().GraderFaceAngle1/angDel;  // e.g. 60
deg
    int f2chunks=0.5*ConfigP->data().GraderFaceAngle2/angDel;  // e.g. 40
deg
    int f3chunks=0.5*ConfigP->data().GraderFaceAngle3/angDel;  // e.g. 50
deg
    int f4chunks=0.5*ConfigP->data().GraderFaceAngle4/angDel;  // e.g. 40
deg

    WVREAL32 f1scores[360];
    WVREAL32 f2scores[360];
    WVREAL32 f3scores[360];
    WVREAL32 f4scores[360];
    WVREAL32 fourFaceScores[360];
    WVREAL32 logScores[360];
    WVREAL32 faceScores[360];
    pu_bzero((WVPOINTER)f1scores,sizeof(f1scores));
```

```
pu_bzero((WVPOINTER)f2scores,sizeof(f2scores));
pu_bzero((WVPOINTER)f3scores,sizeof(f3scores));
pu_bzero((WVPOINTER)f4scores,sizeof(f4scores));

for (scoreIx=0; scoreIx<scoreCount; scoreIx++) {
      int logScore=segmentScores[scoreIx];
      for (ix=1; ix<=f1chunks; ix++) {
            int tix=scoreIx+ix;
            if (tix<0) tix+=scoreCount;
            if (tix>=scoreCount) tix-=scoreCount;
            logScore+=segmentScores[tix];
            tix=scoreIx-ix;
            if (tix<0) tix+=scoreCount;
            if (tix>=scoreCount) tix-=scoreCount;
            logScore+=segmentScores[tix];
            if (ix==f1chunks) f1scores[scoreIx]=logScore;
            if (ix==f2chunks) f2scores[scoreIx]=logScore;
            if (ix==f3chunks) f3scores[scoreIx]=logScore;
            if (ix==f4chunks) f4scores[scoreIx]=logScore;
      }

      // keep track of best single face score.
      if (logScore>maxF1Score) {
            maxF1Ix=scoreIx;
            maxF1Score=logScore;
      }
}

//now compute best opening face angle taking into account all
//for faces.
maxScore=0;
for (ix=0; ix<scoreCount; ix++) {
      float score;
      lix=ix;
      score=f1scores[lix];
      lix=ix+scoreCount/4;
      if (lix>=scoreCount) lix-=scoreCount;
      score+=f2scores[lix];
      lix=ix+scoreCount/2;
      if (lix>=scoreCount) lix-=scoreCount;
      score+=f3scores[lix];
      lix=ix+scoreCount*0.75;
      if (lix>=scoreCount) lix-=scoreCount;
      score+=f4scores[lix];
      if (score>maxScore) {
            maxIx=ix;
            maxScore=score;
      }
      fourFaceScores[ix]=score;
}

// create arrays of scores of size 360 for convenient display.
// save to disk.
for (ix=0; ix<360; ix++ ) {
      faceScores[ix]=f1scores[ix/angDel];
      logScores[ix]=fourFaceScores[ix/angDel];
}

int OutFd=pu_open("$(DATADIR)/temp/facescores",
      PU_O_WRONLY|PU_O_CREAT|PU_O_APPEND,0666);
```

```
    if (OutFd) {
        pu_write(OutFd,(WVPOINTER)faceScores,360*sizeof(INT32));
        pu_close(OutFd);
    }
    OutFd=pu_open("$(DATADIR)/temp/logscores",
        PU_O_WRONLY|PU_O_CREAT|PU_O_APPEND,0666);
    if (OutFd) {
        pu_write(OutFd,(WVPOINTER)logScores,360*sizeof(INT32));
        pu_close(OutFd);
    }

    bestAngle=180-maxIx*angDel;
    if (bestAngle<0) bestAngle+=360;
    printf("Best angle at %i (%.0f) score=%i",
        maxIx*angDel,bestAngle,maxScore);
    myClist->RefAngle(bestAngle*D2R);
}


////////////////////////////////////////////////////////////////////////
////
// This method groups segments groups of contiguous values greater than
<minVal>
// in an image.  Contiguous points are grouped together into objects,
// which we call defects here.  When done, those groups (defects) that
// contain fewer than <minCount> pixels are removed from the image.
//
// This is a single pass segmentor.
int Img::SegmentImage( int minVal, int minCount, List<Defect>* dListP ){

    // Create a blank array that is the same size as the input image.
This
    // array will hold tags which indicate which group a pixel belongs
to.
    int imgBytes=width*height*sizeof(BDINT);
    BDINT *targetImg=(BDINT *)pu_calloc(imgBytes,1);

    // Create an array of defect objects larger than the expected
number of
    // defects (of course, dynamic sizing would be more robust).
    // The WLE (Walter's length encoding) object is a generic run
length encoding
    // object that describes a group of pixels as a set of 'runs' of
    // consecutive points.  WLE_P is a pointer to a WLE object.

    static int MAXTAGS=1000;
    int counts[maxtag];    // the number of points with each tag number
    WLE_P wles[maxtag];    // description of all of the points in each
object

    // initalize
    pu_bzero((WVPOINTER)wles,sizeof(wles));
    pu_bzero((WVPOINTER)counts,sizeof(counts));

    int nTags=0;           // current tag number
    int isIn=0;            // flag indicating if preceding points are
inside

    int runStartX,runEndX,tagit,lasttag=0,mytag;
    int iy,ix;

    // pointers to the input data, and the copy containg the tags
```

```
        BDINT *dataP=data;
        BDINT *dP=dataP;
        BDINT *tP=targetImg;
        int it;

        // traverse the top line of the image, looking for consecutive
pixels.
        // greater than <minVal> (call these target pixels from now on).
        // when a new target pixel is found, assign it a new tag number.
For
        // subsequent target pixels, assign it the tag for the current
run,
        // so all consecutive pixels have the same unique tag.
      for (ix=0; ix<width;ix++){
            if (dP[ix]>minVal) {
                  // We have found a target pixel.
                  if (!isIn){
                        // the first target pixel from the beginning or
after non
                        // target pixel.  Get a new tag number.
                        nTags++;
                        runStartX=ix;
                  }
                  // now inside a run.  Set isIn flag, and assign the
pixel value
                  // in the target image the current tag  number.
                  isIn=TRUE;
                  tP[ix]=nTags;
                  counts[nTags]++;
            } else {
                  // The current pixel is not a target pixel.  Put a
zero in the
                  // target image.
                  tP[ix]=0;
                  if (isIn) {
                        // The previous pixel was part of a run (one or
more) of target
                        // pixels.  Create a new wle for this run, and
add the run of
                        // target pixels to the wle.
                        wles[nTags]=new Wle();
                        wles[nTags]->StartEncoding(20);
                        wles[nTags]->AddRunSize(runStartX,0,ix-
runStartX);
                  }
                  isIn=FALSE;
            }
      }

        // Reached the end of the line.  If the last pixel is part of a
run of target
        // pixels, Create a new wle and add the run to it.

      if (isIn) {
            wles[nTags]=new Wle();
            wles[nTags]->StartEncoding(20);
            wles[nTags]->AddRunSize(runStartX,0,ix-runStartX+1);
      }

        // Repeat the above procedure for subsequent lines.  This time for
each new
```

```
        // target pixel, look at the point above it to see if it is
    tagged.
        for (iy=1; iy<height; iy++) {
                tagit=FALSE;
                isIn=0;
                dP=&dataP[iy*width];
                tP=&targetImg[iy*width];
                int tagIsAdopted=0;

                // start from the beginning of the line and look for target
    pixels
                for (ix=0; ix<width;ix++){
                      if (dP[ix]>minVal){
                            // the point is a target pixel.
                            if (!isIn) {
                                  // This is the first of a run of target
    pixels.  Get a new
                                  // tag id, and record where this run
    started.
                                  lasttag=nTags;
                                  if (nTags<maxtag) nTags++;
                                  mytag=nTags;
                                  runStartX=ix;
                                  isIn=TRUE;
                            }
                            // for the first and target subsequent pixels,
    Look at the pixel
                            // above it.  If it is tagged, adopt the tag of
    the pixel above it,
                            // and put the current tag bag on the stack
                            int newTag=tP[ix-width];
                            if (newTag) {
                                  // The pixel above the current pixel has a
    tag.  Have we
                                  // alfready adopted a tag?
                                  if (tagIsAdopted) {
                                        // The tag has already been adopted,
    so we are already
                                        // part of an object.  The new tag
    belongs to an object that
                                        // has been separate so far, but is
    now joined from below
                                        // by our current object.  Merge the
    two together, and delete
                                        // the new object. (We can do a
    quick merge because we know
                                        // that there are no overlapping
    points).
                                        wles[mytag]-
    >QuickMerge(*wles[newTag]);
                                        delete wles[newTag];
                                        wles[newTag]=NULL;
                                  } else {
                                        nTags=lasttag;
                                        mytag=tP[ix-width];
                                  }
                            }
                            runEndX=ix;
                            // Set flag to encode the run if this is last
    pixel in the line.
                            if (ix==width-1) tagit=TRUE;
```

```
                    } else {
                        // The point is not a target pixel. Set flag to
encode previous run,
                        if (isIn) {
                            tagit=TRUE;
                            isIn=FALSE;
                        }
                    }
                    if (tagit) {
                        // End of the line, or end of run of target
pixels.
                        // If this is a new tag, create a new wle amd
add the new run
                        // to it.  If it is an adopted tag, add the run
to the wle for
                        // the adopted wle.
                        tagIsAdopted=0;
                        if (!wles[mytag]) wles[mytag]=new Wle();
                        wles[mytag]->AddRunSize(runStartX,iy,runEndX-
runStartX+1);


                        // set all of the values in the run to the tag
number.
                        for (;runStartX<=runEndX; runStartX++){
                            tP[runStartX]=mytag;
                            counts[mytag]++;
                        }
                        tagit=FALSE;
                    }
                }
            }

        // Browse through all of the wles that have been identified, and
remove
        // the wles for which there are not enough points (number of
points < <minCount>)
        int ndefects=0,nth=0;
        int maxArea=0,maxix,npts;
        for (it=1; it<=nTags; it++) {
            if (wles[it] ) {
                npts=wles[it]->Area();
                if (npts>maxArea) {
                    maxArea=npts;
                    maxix=it;
                }
                // The wle exists, which means it was not merged with
another one.
                // are there enough points in the wle?  Ignore defects
that occur only
                // on first and last slices.
                if ( (wles[it]->Area()>=minCount) &&
                    (wles[it]->Left()<width-1) &&
                    (wles[it]->Right()>0))                {
                    ndefects++;
                    // Add defect to the defects list
                    // (get rid of defects only on first and last
slices)
                    Defect defect =
Defect(wles[it],TYPE_UNKNOWN,nth++);
                    dListP->Add(defect);
```

```
                        } else {
                              // not enough points in the wle.  Zero out the
    points in the
                              // original image.
                              WRUNVAL run;
                              wles[it]->FirstRunVal(&run);
                              while (run.l>=0) {
                                    dP=&dataP[run.y*width+run.x];
                                    for (ix=0; ix<run.l; ix++) {
                                          *dP++=0;
                                    }
                                    wles[it]->NextRunVal(&run);
                              }
                        }

                        delete wles[it];
                  }
            }
            if (targetImg) pu_free((WVPOINTER)targetImg);
            return(ndefects);

      }


      ////////////////////////////////////////////////////////////////////////
      ////
      // Analyze each defect in the defect list to find knot type and knot
      depth.
      // dpeth is a minimum and maximum radius (distance from center of log)
      of
      // the defect.
      void vTree::AnalyzeDefectList(){
            int k,r,zmin,zmax,zmean,val,val0,val1,val2,zout,dAngle;
            int meanVal,done,isin,rmin,rmax,rc,nbOut,nbIn,rminDone;
            double
      meanAngle,cosAmin,cosAmax,cosAmean,sinAmin,sinAmax,sinAmean;
            double cos1,cos2,sin1,sin2,lineSum,neighborSum;
            int NEIGHBOR_OFFSET=dConfigP->data().NeighborOffset;
            int MIN_NB_PIX=dConfigP->data().MinNbDefectPixels;
            int SEARCH_RADIUS = dConfigP-
      >data().MaximalLogRadiusCM*10/MmPerPix();
            int HEART_WOOD_RADIUS = dConfigP-
      >data().HeartWoodRadiusCM*10/MmPerPix();
            int bigKnotLen=dConfigP->data().BigKnotRadialLengthMM/MmPerPix();
            int bigKnotAngle=dConfigP->data().BigKnotAngle;
            WVUINT16* meanLine =
      (WVUINT16*)pu_malloc(SEARCH_RADIUS*sizeof(WVUINT16));

            // Browse through all defects
            Defect* pDefect=DefectsList.First();
            while (pDefect){

                  // Precompute defect area variables
                  zmin=pDefect->ZMin();
                  zmax=pDefect->ZMax();    if (zmax>=NSlices())
      zmax=NSlices()-1;
                  zmean=(zmin+zmax)/2;
                  dAngle=(pDefect->AngleMax()-pDefect->AngleMin())/D2R;
                  meanAngle = (pDefect->AngleMax()+pDefect->AngleMin())/2;
                  cosAmin=cos(pDefect->AngleMin());
```

```
                cosAmax=cos(pDefect->AngleMax());
                sinAmin=sin(pDefect->AngleMin());
                sinAmax=sin(pDefect->AngleMax());
                cosAmean=cos(meanAngle);
                sinAmean=sin(meanAngle);
                cos1=(cosAmean+2*cosAmin)/3;
                sin1=(sinAmean+2*sinAmin)/3;
                cos2=(cosAmean+2*cosAmax)/3;
                sin2=(sinAmean+2*sinAmax)/3;

                // Get neighbor non defect slice at zout
                zout=zmax+NEIGHBOR_OFFSET;
                if (zout>=NSlices()) {
                      zout=zmin-NEIGHBOR_OFFSET;
                      if (zout<0) zout=0;
                }
                // Compare sum of pixels along 3 lines on defect slice with
                // the one on non defect neighbor slice to find defect type
                lineSum=neighborSum=0;
                for (r=0/*ConfigP-
>data().InnerRadiusCM*/;r<HEART_WOOD_RADIUS;r++){
                      neighborSum += imgPs[zout].CtVal(imgPs[zout].Pithx() +
r*cosAmean,
                            imgPs[zout].Pithy() + r*sinAmean)
                            + imgPs[zout].CtVal(imgPs[zout].Pithx() +
r*cos1,
                            imgPs[zout].Pithy() + r*sin1)
                            + imgPs[zout].CtVal(imgPs[zout].Pithx() +
r*cos2,
                            imgPs[zout].Pithy() + r*sin2);

                      lineSum += imgPs[zmean].CtVal(imgPs[zmean].Pithx() +
r*cosAmean,
                            imgPs[zmean].Pithy() + r*sinAmean)
                            + imgPs[zmean].CtVal(imgPs[zmean].Pithx() +
r*cos1,
                            imgPs[zmean].Pithy() + r*sin1)
                            + imgPs[zmean].CtVal(imgPs[zmean].Pithx() +
r*cos2,
                            imgPs[zmean].Pithy() + r*sin2);
                }
                if (lineSum < neighborSum){
                      pDefect->SetType(TYPE_VOID);
                      for (r=0;r<SEARCH_RADIUS;r++) meanLine[r]=2000;
                }
                else {
                      pDefect->SetType(TYPE_KNOT);
                      for (r=0;r<SEARCH_RADIUS;r++) meanLine[r]=0;
                }

                // Store the average CT value on non defect neighbor slice
                // This value will be used as threshold to identify the
defect area
                meanVal=neighborSum/(3*HEART_WOOD_RADIUS);

                // Build radial line from 3 different radial lines on
defect's CT slices
                for (k=zmin; k<=zmax; k++){
                      for (r=0;r<SEARCH_RADIUS;r++){
                            val0=imgPs[k].CtVal(imgPs[k].Pithx() +
r*cosAmean,
```

```
                                imgPs[k].Pithy() + r*sinAmean);
                          val1=imgPs[k].CtVal(imgPs[k].Pithx() + r*cos1,
                                imgPs[k].Pithy() + r*sin1);

                          val2=imgPs[k].CtVal(imgPs[k].Pithx() + r*cos2,
                                imgPs[k].Pithy() + r*sin2);
                          if (pDefect->Type()==TYPE_VOID) {
                                // Get min value
                                val = min(min(val0,val1),val2);
                                meanLine[r]=min(meanLine[r],val);
                          }
                          else {
                                // Get max value
                                val = max(max(val0,val1),val2);
                                meanLine[r]=max(meanLine[r],val);
                          }
                    }
              }
              // Go along radial line values to find and store defect
radial limits
              rc=done=rminDone=isin=rmin=nbOut=nbIn=0;
              rmax=SEARCH_RADIUS;
              if (dConfigP->data().DetectDefectRadialLimits){
                    while (!done && (rc < SEARCH_RADIUS)){
                          if (IsDefect(meanLine[rc],meanVal,pDefect-
>Type())){
                                // First point inside this defect part
                                if (!isin){
                                      isin=1;
                                      nbOut=0;
                                      if (!rminDone){
                                            nbIn=1;
                                            //rmin=rc;
                                            rmin=0; // Assume rmin=0 for
all knots
                                      }
                                }
                                // Next point inside defect
                                else {

                                      nbIn++;
                                }
                          }
                          else {
                                // First point outside defect
                                if (isin){
                                      isin=0;
                                      if (!rminDone && (nbIn >
MIN_NB_PIX)){

                                            rminDone=1;
                                            pDefect->SetRMin(rmin);
                                      }
                                      if (rminDone){
                                            rmax=rc;
                                            nbOut=1;
                                      }
                                }
                                // Next point outside defect
                                else if (rminDone){
                                      nbOut++;
                                      if (nbOut > MIN_NB_PIX){
                                            done=1;
```

```
                                                pDefect->SetRMax(rmax);
                                            }
                                        }
                                    }
                                    rc++;
                                }
                            }

                            // Enlarge too small radial size
                            if (rmax-rmin<dConfigP->data().DefectMinRadLen) {
                                sprintf( logMsg, "Defect radial length too small (Nth
%i): rmin=%i rmax=%i (zmin=%i zmax=%i)",(int)pDefect-
>NTh(),rmin,rmax,(int)pDefect->ZMin(),(int)pDefect->ZMax());
                                LogEntry(  SMF_DEBUG_LEVEL_5,
                                    SOURCE_LVT_FIRST,
                                    LVT_DEBUG_MSG,
                                    TEXTTYPE_US_ASCII,
                                    ( unsigned char * )logMsg,
                                    logFile,
                                    true
                                    );
                            }

                            while (rmax-rmin<dConfigP->data().DefectMinRadLen){
                                if ((rmax==SEARCH_RADIUS) || (rmin==0))
                                    break;
                                rmax++;
                                rmin--;
                            }

                            // If defect is a knot, set knot type
                            if (pDefect->Type()==TYPE_KNOT){
                                if (((rmax-rmin)>bigKnotLen) && (dAngle>bigKnotAngle))
                                    pDefect->SetKnotType(BIG_KNOT);
                                else
                                    pDefect->SetKnotType(SMALL_KNOT);
                            }

                                // set knot flag in ct image data
                                int zmax = (pDefect->ZMax() < NSlices())? pDefect-
>ZMax():NSlices()-1;
                                for (int z=pDefect->ZMin();z<=zmax;z++){
                                    imgPs[z].SetKnotArea(pDefect->RMin(),pDefect-
>RMax(),
                                        pDefect->AngleMin(),pDefect->AngleMax());
                                }
                            }

                            // Go to next defect
                            pDefect = DefectsList.Next();
                        }
                        pu_free((WVPOINTER)meanLine);
                    }


                    ////////////////////////////////////////////////////////////////////
                    ////
                    //  Draw a cut on the radial view.  A sawn face of a log can be thought
                    // of as a cord of a circle; this cord subtends an angle.  Here we draw
                    // to lines to indicate the two angles where the chord line intersects
                    the
```

```
// circle
void RView::DrawCurrentCut(){

      // Erase previously drawn lines
      if (cutsDrawn) {
            EraseArea(cut1Area);
            EraseArea(cut2Area);
      }

      int clr=cLut[CLR_STEEL_BLUE];

      // rotation angle of the log
    int rotationAngle=180-myClist->RefAngle()/D2R;

      // distance of cut from center of log
    double xx=myClist->Distance(0);

    double radius=150 /*radii[rotationAngle]*/;

      // compute angle subtended by chord at distance xx
    double subtendedAngle=acos(xx/radius)/D2R;

      // draw line at position (rotationAngle-subtendedAngle/2)
    rotationAngle -=subtendedAngle/2;
    while(rotationAngle<0) rotationAngle+=360;
    while(rotationAngle>359) rotationAngle-=360;

      // scale to size of radial image
    int yy=rotationAngle*height/(double)360;

      // define drawing area and draw lines into the image
      cutsDrawn=true;
      cut1Area.left=0;
      cut1Area.right=width;
      cut1Area.top=yy-1;
      cut1Area.bottom=yy+1;
      if (yy<=0) {cut1Area.top++; cut1Area.bottom++;};
      if (yy>=height) {cut1Area.top--; cut1Area.bottom--;}
    DrawPixLine(clr,10,yy,Width()-1,yy);
      RedrawArea(cut1Area);

      // repeat the process for the other side of the cut face
    rotationAngle +=2*subtendedAngle;
    while(rotationAngle>359) rotationAngle-=360;
    yy=rotationAngle*height/(double)360;
      cut2Area.left=0;
      cut2Area.right=width;
      cut2Area.top=yy-1;
      cut2Area.bottom=yy+1;
      if (yy<=0) {cut2Area.top++; cut2Area.bottom++;};
      if (yy>=height) {cut2Area.top--; cut2Area.bottom--;}
    DrawPixLine(clr,10,yy,Width()-1,yy);
      RedrawArea(cut2Area);

      // redraw the line indicating position of currently selected ct
slice
      DrawSliceLine();

}
```

```
////////////////////////////////////////////////////////////////////////
////
// Find the center of the annual rings (growth center) of each CT image.
// Then smooth the position over the length of the tree, rejecting
// points of low confidence and points that differ significantly from
the
// local average
void vTree::FollowPith() {
    if (pithComputed) return;
      pithComputed=TRUE;
    int slice;
    float *tpithx,*tpithy;

      // create status bar showing progress.
      createWVProgressWindow("Finding Growth Centers:", 0, NSlices(),
1);
      for (slice=0; slice<NSlices(); slice++) {
          double xx,yy;
          imgPs[slice].FindPith(xx,yy);
          updateWVProgressWindow();
      }
      endWVProgressWindow();

    // Smooth out the detected pith positions by making running average.
    // Throw out outlyers and re-average.
    if (ConfigP->data().SmoothPith) {
          tpithx=(float*)pu_malloc(NSlices()*sizeof(float));
          tpithy=(float*)pu_malloc(NSlices()*sizeof(float));

          // for each slice not near the ends
          for (slice=5; slice<NSlices()-5; slice++) {
              int i;
              float sumx=0;
              float sumy=0;
              float mxdx=0;
              float tx,ty,dx,dy,mx,my;
              float lastx,lasty;    //previous x,y of pith
              float cx,cy;
              int lastConfidence=0;

              // find the average pith position for the nearest 11
slices
              for (i=slice-5; i<=slice+5; i++) {
                  cx=imgPs[i].Pithx();
                  cy=imgPs[i].Pithy();
                  //if the pith confidence is low, use the
previous slice
                  // if there was confidence in that slice.
                  if ((imgPs[i].PithConfidence() ==0) &&
lastConfidence) {
                      sumx+=lastx;
                      sumy+=lasty;
                  } else {
                      sumx+=cx;
                      sumy+=cy;
                  }
                  lastx=cx;
                  lasty=cy;
                  lastConfidence=imgPs[i].PithConfidence();
              }
              tx=sumx/11.0;
              ty=sumy/11.0;
```

```
                    // of the nearest 9 slices, find the one with the
greatest difference
                    // from the running average, and remove it from the
average position.
                    for (i=slice-4; i<=slice+4; i++) {
                        dx=ABS(imgPs[i].Pithx()-tx);
                        dy=ABS(imgPs[i].Pithy()-ty);
                        if (dx*dy>=mxdx) {
                            mxdx=dx*dy;;
                            mx=imgPs[i].Pithx();
                            my=imgPs[i].Pithy();
                        }
                    }
                    sumx-=mx;
                    sumy-=my;
                    tpithx[slice]=(sumx + 0.5)/10.0;
                    tpithy[slice]=(sumy + 0.5)/10.0;
                }

                // use the first averaged position for the first and last
slices.
                for(slice=0; slice<5; slice++) {
                    tpithx[slice]=tpithx[5];
                    tpithy[slice]=tpithy[5];
                    tpithx[NSlices()-1-slice] = tpithx[NSlices()-6];
                    tpithy[NSlices()-1-slice] = tpithy[NSlices()-6];
                }

                // replace the original pith coordinates with the smoothed
coordinates.
                for (slice=0; slice<NSlices(); slice++) {
                    imgPs[slice].SetPith(tpithx[slice],tpithy[slice]);
                }
                pu_free((WVPOINTER)tpithx);
                pu_free((WVPOINTER)tpithy);
        }
    }

    ////////////////////////////////////////////////////////////////////////
    ////
    // Find position of pith
    // Find geometric center, then find maximum X and Y average gradients
    void CtImg::FindPith(double &xorig,double& yorig) {
        int ix,iy,cx,cy,ind;
        int foundit=0;
        CtImg img;
        INT nvals=0;
        float geomCentX=0,geomCentY=0;

        geomCentX=0;
        geomCentY=0;
        // Find approximate geometric center of cross section of CT slice.
          // Values greater than 150 have been removed, and extraneous
points on
         // the outside of the log have been removed.
          // Save time by analyzing evegeomCentY 10th line of the image.
        for (iy=0; iy<Height(); iy+=10) {
            for (ix=0; ix<Width(); ix++) {
                ind = ix + Width()*iy;
                int val=CtVal(ind);
```

```
                    if ((val>150)) {
                        nvals++;
                        geomCentX+=ix;
                        geomCentY+=iy;
                    }
                }
        }
        if (nvals) {
                geomCentX /= nvals;
                geomCentY /= nvals;
        } else {
                geomCentX=Width()/2;
                geomCentY=Height()/2;
        }
        xorig=geomCentX;
        yorig=geomCentY;
        cy=geomCentY;
        cx=geomCentX;

        //Attempt to find center of annual rings.   The idea is to find
        // the absolute values of the x and y gradient of each point near
        // the geometric center.  The y position with the highest X
    gradients
        // (hopefully the place where we cross the highest number of annual
        // rings) should be the pith.  Same idea for the X position with the
        // highest Y gradients.
        if (ConfigP->data().PithDistance) {

                // define a square region centered around the geometric
    center, of
                // a specified size.
                int dist=ConfigP->data().PithDistance;  //e.g. 100 pixels =
    8cm
                int xs=cx-dist/2;   //xs,ys: upper left coordinates
                int ys=cy-dist/2;
                if (xs<0) xs=1;
                if (ys<0) ys=1;
                int xe=xs+dist;     //xe,ye: lower right coordinates
                int ye=ys+dist;
                if (xe>Width()-2) xe=Width()-2;
                if (ye>Width()-2) ye=Height()-2;

                int index;
                int cindex=0;
                float val,sum, xmax=0,ymax=0;
                int yind=0, xind=0;


                // For each horizontal line in the square, find the gradient
    and add
                // the absolute value of the gradient to an accumulator
                for (iy=ys; iy<=ye; iy++) {
                        sum=0;
                        index=iy*Width()+xs;
                        for (ix=xs; ix<=xe; ix++) {
                                val=CtVal(index-1)-CtVal(index);
                                if (val<0) val=-val;
                                // remove points with large gradients that are
    probably not caused
                                // by growth rings.
                                if (val<ConfigP->data().MaxPithGradient)
                                        sum += val;
```

```
                        index++;
                    }
                    // record the highest sum for a line
                    if (sum>ymax) {
                        ymax=sum;
                        yind=iy;
                    }

                }

                // likewise, find the gradients of vertical lines and find
    the
                // sum of the absolute values of the gradient
                for (ix=xs; ix<=xe; ix++) {
                    index=ix+ys*Width();
                    sum=0;
                    for (iy=ys; iy<=ye; iy++){
                        val=CtVal(index)-CtVal(index+Width());
                        if (val<0) val=-val;
                        if (val<ConfigP->data().MaxPithGradient)
                            sum +=val;
                        index+=Width();
                    }
                    if (sum>xmax) {
                        xmax=sum;
                        xind=ix;
                    }
                }

                // if the detected position is on the edge of our square,
                // indicate lack of confidence in the result.
                if ((xind<=xs) || (xind>=xe) || (yind<=ys) || (yind>=ye)) {
                    pithConfidence=0;
                } else {
                    pithConfidence=1;
                }
                xorig=xind;
                yorig=yind;

        }
        pithx=xorig;
        pithy=yorig;

        if (pithx < 0 || pithy < 0 ) {
            MessageBox( mainWindow, "Failed to find valid pith", "Pith
    Locate Error", MB_SYSTEMMODAL | MB_OK | MB_ICONEXCLAMATION);
        }
        return;
    }


    ////////////////////////////////////////////////////////////////////////
    ////
    //  Draw a cut on the radial view.  A sawn face of a log can be thought
    // of as a cord of a circle; this cord subtends an angle.  Here we draw
    // to lines to indicate the two angles where the chord line intersects
    the
    // circle
    void RView::DrawCurrentCut(){

        // Erase previously drawn lines
```

```
        if (cutsDrawn) {
              EraseArea(cut1Area);
              EraseArea(cut2Area);
        }

        int clr=cLut[CLR_STEEL_BLUE];

        // rotation angle of the log
      int rotationAngle=180-myClist->RefAngle()/D2R;

        // distance of cut from center of log
      double xx=myClist->Distance(0);

      double radius=GetRadius(myClist->RefAngle());

        // compute angle subtended by chord at distance xx
      double subtendedAngle=acos(xx/radius)/D2R;

        // draw line at position (rotationAngle-subtendedAngle/2)
      rotationAngle -=subtendedAngle/2;
      while(rotationAngle<0) rotationAngle+=360;
      while(rotationAngle>359) rotationAngle-=360;

        // scale to size of radial image
      int yy=rotationAngle*height/(double)360;

        // define drawing area and draw lines into the image
        cutsDrawn=true;
        cut1Area.left=0;
        cut1Area.right=width;
        cut1Area.top=yy-1;
        cut1Area.bottom=yy+1;
        if (yy<=0) {cut1Area.top++; cut1Area.bottom++;};
        if (yy>=height) {cut1Area.top--; cut1Area.bottom--;}
      DrawPixLine(clr,10,yy,Width()-1,yy);
        RedrawArea(cut1Area);

        // repeat the process for the other side of the cut face
      rotationAngle +=2*subtendedAngle;
      while(rotationAngle>359) rotationAngle-=360;
      yy=rotationAngle*height/(double)360;
        cut2Area.left=0;
        cut2Area.right=width;
        cut2Area.top=yy-1;
        cut2Area.bottom=yy+1;
        if (yy<=0) {cut2Area.top++; cut2Area.bottom++;};
        if (yy>=height) {cut2Area.top--; cut2Area.bottom--;}
      DrawPixLine(clr,10,yy,Width()-1,yy);
        RedrawArea(cut2Area);

        // redraw the line indicating position of currently selected ct
slice
        DrawSliceLine();

}
```

Although the invention has been described with reference to particular embodiments, the description is only an example of the invention's application and should not be taken as a limitation. Various adaptations and combinations of features of the

embodiments disclosed are within the scope of the invention as defined by the following claims.

CLAIMS

I claim:

1. A defect structure for a log, comprising:

a two-dimensional array of data values, the data values corresponding to respective values of a first coordinate Z indicating distance along the log and a second coordinate θ indicating an angle around the log, wherein

each data value indicates a property of the log that is evaluated along a ray that originates at a center point corresponding to the value of the first coordinate Z for the data value and extends in a direction corresponding to the value of the second coordinate θ for the data value.

2. The defect structure of claim 1, wherein for each value of the first coordinate Z, the center point corresponding to that value of the first coordinate Z is a growth center of the log.

3. The defect structure of claim 2, wherein for each data value, the ray evaluated to determine the data value is directed at an upward angle along the log, the upward angle being characteristic of the log.

4. The defect structure of claim 3, wherein the upward angle is a growth direction of tree limbs and is characteristic of trees of a species that produced the log.

5. The defect structure of claim 1, wherein for each data value, the ray evaluated to determine the data value is directed at an upward angle along the log, the upward angle being characteristic of the log.

6. The defect structure of claim 5, wherein the upward angle is a growth direction of tree limbs and is characteristic of trees of a species that produced the log.

7. The defect structure of claim 1, wherein each data value indicates a property of the log that is only evaluated in a range of distances along the ray that originates at the center point corresponding to the values of the first and second coordinates for the data value.

8. The defect structure of claim 7, wherein the range extends from a first distance from the center point to a second distance from an edge of the log, the first distance excluding core wood from the range, the second distance excluding bark from the range.

9. The defect structure of claim 1, where each data value indicates presence or absence of a defect at a point in the log corresponding to the values of the first and second coordinates for the data value.

10. The defect structure of claim 9, wherein when one of the data values indicates the presence of a defect, the data value also indicates a location of the defect along the ray evaluated for the data value.

11. A method for generating a description of a log, comprising:

for a set of locations along the length of the log, finding at each location a center point of the log;

for a set of rays that extend from the center points, evaluating a property of the log along each ray to generate a data value corresponding to a value of a first coordinates Z identifying the location for the ray and a value of a second coordinate θ identifying a direction of the ray; and

constructing a two-dimensional data structure that describes the log, the two-dimensional data structure including the data values at positions in the two-dimensional data structure according to the respective values of the first and second coordinates.

12. The method of claim 11, wherein finding the center point for one of the locations comprises finding a center of annual growth rings in the log at the location.

13. The method of claim 11, wherein evaluating the property of the log along one of the rays comprises evaluating density of the log along the ray to determine whether there is a defect along the ray, in the log.

14. The method of claim 13, further comprising performing CT scanning of the log to generate a three-dimensional data structure that provides the densities evaluated along the rays.

15. A system for evaluating a log, comprising a program code that is computer executable for manipulating a data structure for the log, wherein the data structure comprises:

a two-dimensional array of data values, the data values corresponding to respective values of a first coordinate Z indicating distance along the log and a second coordinate θ indicating an angle around the log, wherein

each data value indicates a property of the log that is evaluated along a ray that originates at a center point corresponding to the value of the first coordinate Z for the data value and extends in a direction corresponding to the value of the second coordinate θ for the data value.

16. The system of claim 15, further comprising:

a display device; and

a processor capable of executing the program code, wherein in executing the program code the processor controls display of an image on the display device, the image including pixels that correspond to the data values of the data structure and have shades defined by the respective data values.

17. The system of claim 16, wherein executing the program code further superimposes marks in the image, the marks indicating boundaries of one or more faces of the log that results when sawing the log.

18. The system of claim 17, wherein executing the program code further permits user controlled shifting of the marks relative to the image.

19. The system of claim 15, wherein the program code manipulates the data structure to generate a grade value for the log.

20. A method for grading a log, comprising:

determining a computer tomography (CT) data structure representing the log; and

processing the CT data structure to arrive at a grade for the log.

21. The method of claim 20, wherein processing the CT data comprises:

creating a two-dimensional array of data values indicating a defect structure of the log, each data value indicating presence or absence of a defect in a portion of the log corresponding to the data value, wherein each data value in the two dimensional array corresponds to values of a fist coordinate Z indicating a position along the length of the log and a second coordinate θ indicating an angle around the log;

evaluating the two-dimensional array to determine sizes of blocks in the two-dimensional array, that are free of data values indicating defects; and

assigning the grade to the log according to the sizes.

22. The method of claim 21, wherein evaluating the two-dimensional array comprises:

creating an image having pixels that correspond to the data values and have shades according to the data values; and

viewing of the image by a grader, wherein the grader recognizes the sizes of the blocks through evaluating areas of the image having a shade indicating an absence of defects.

23. The method of claim 22, wherein the grader assigns the grade to the log qualitatively based on the viewing of the image.

24. The method of claim 22, further comprising superimposing on the image, marks indicating boundaries of one or more faces of the log that results from a sawing strategy for the log.

25. The method of claim 24, further comprising shifting the image relative to the marks to minimizes defects within the boundaries of the one or more faces.

26. The method of claim 25, further comprising selecting an orientation of the log according to positions of the marks that minimize defects within the boundaries of the one or more faces.

27. The method of claim 21, wherein evaluating the two-dimensional array comprises executing a computer program that manipulates the two-dimensional array.

28. The method of claim 27, wherein executing the computer program comprises:

(a) determining a number N of data values that are consecutive in a direction of the second coordinate θ and correspond to a desired width of defect-free wood;

(b) scanning the two-dimensional array in the direction of the second coordinate until identifying N consecutive data values that indicate absence of a defect;

(c) scanning the two-dimensional array in a direction of the first coordinate to determine a size of a block that is defect free;

(d) increasing the grade value for the log by an amount corresponding to the size of the block that is defect-free;

(e) repeating steps b, c, and d to account for defect-free blocks in the two-dimensional data structure.

29. A method for grading a log comprising:

(a) creating a two-dimensional array of data values indicating a defect structure of the log, each data value indicating presence or absence of a defect in a portion of the log corresponding to the data value, wherein data values in the two dimensional array correspond to values of a fist coordinate Z indicating a position along the length of the log and a second coordinate θ indicating an angle around the log;

(b) selecting a sawing strategy for the log;

(c) selecting an orientation of the log for the sawing strategy;

(d) identifying a sub-array of the two-dimensional array, the sub-array corresponding to a face of the log resulting from the sawing strategy and the orientation;

(e) evaluating the sub-array to determine sizes of blocks in the sub-array, that are free of data values indicating defects; and

(f) assigning a first grade value to the face according to the sizes;

(g) repeating steps (d), (e), and (f) for one or more faces of the log resulting from the sawing strategy and the orientation;

(h) combining the first grade values of the faces to generate a second grade value for the orientation; and

(i) repeating steps (c) to (h) for one or more additional orientations of the log;

(j) assigning a third grade value to the log based on a best of the second grade values.

30. A method for identifying a growth center of a log, comprising:

for each line in a first set of lines through a cross-section of the log, determining an accumulated absolute value of a gradient of density of the log along the line;

identifying a first line that is in the first set and has an accumulated absolute value as large as any determined for lines in the first set;

for each line in a second set of lines through the cross-section of the log, determining an accumulated absolute value of a gradient of density of the log along the line;

identifying a second line that is in the second set and has an accumulated absolute value as large as any determined for lines in the second set; and

identifying the growth center as being at an intersection of the first line and the second line.

31. The method of claim 30, wherein the lines in the first set are perpendicular to the lines in the second set.

32. The method of claim 30, further comprising:

determining a geometric center of the log; and

selecting a central area that is smaller than the cross-section of the log and contains the geometric center of the log, wherein each line in the first and second sets passes through the central area.

33. The method of claim 32, wherein determining the accumulated absolute values of the gradient of the density of the log only considers densities for points inside the central area.

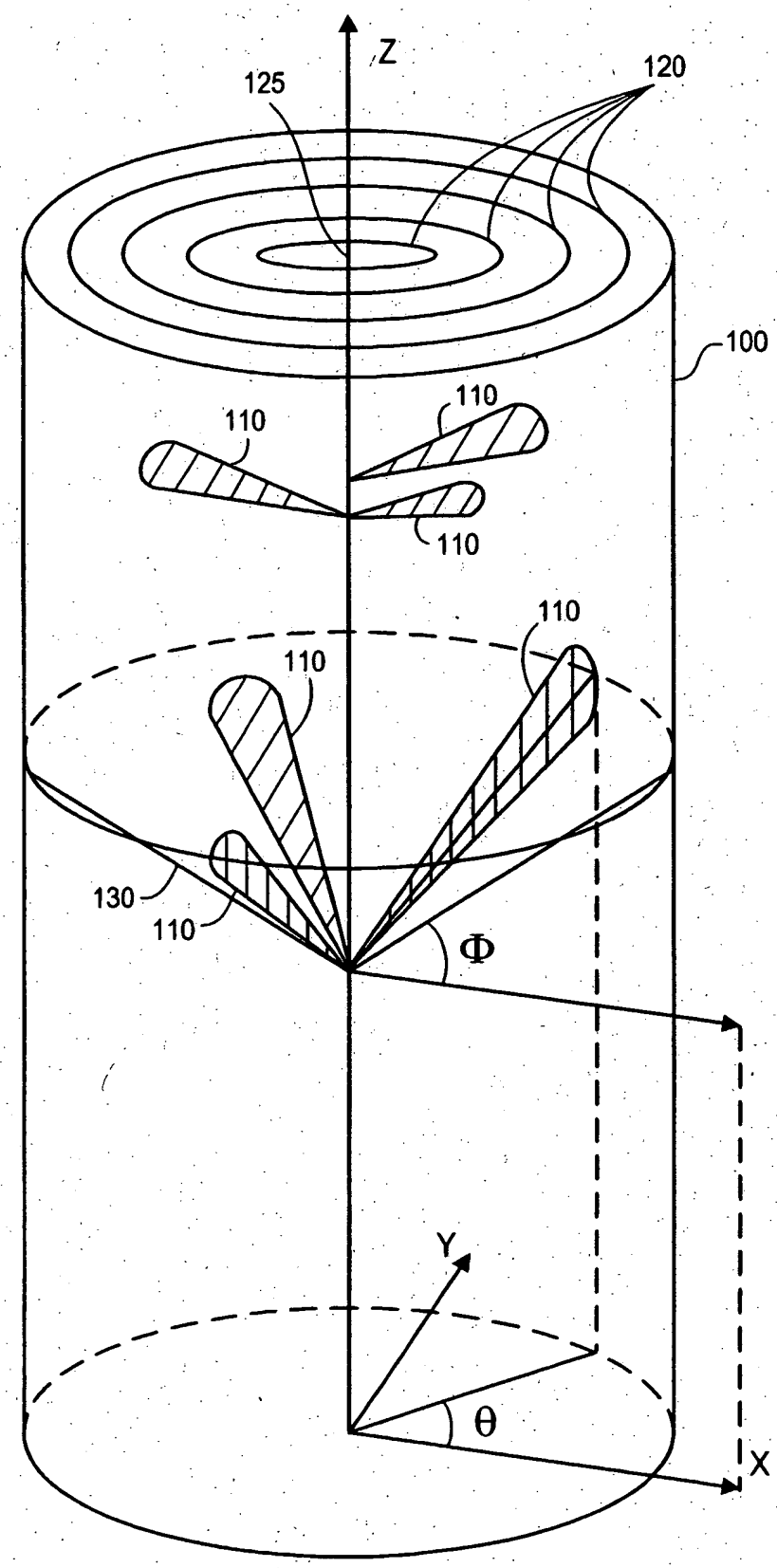


Fig 1

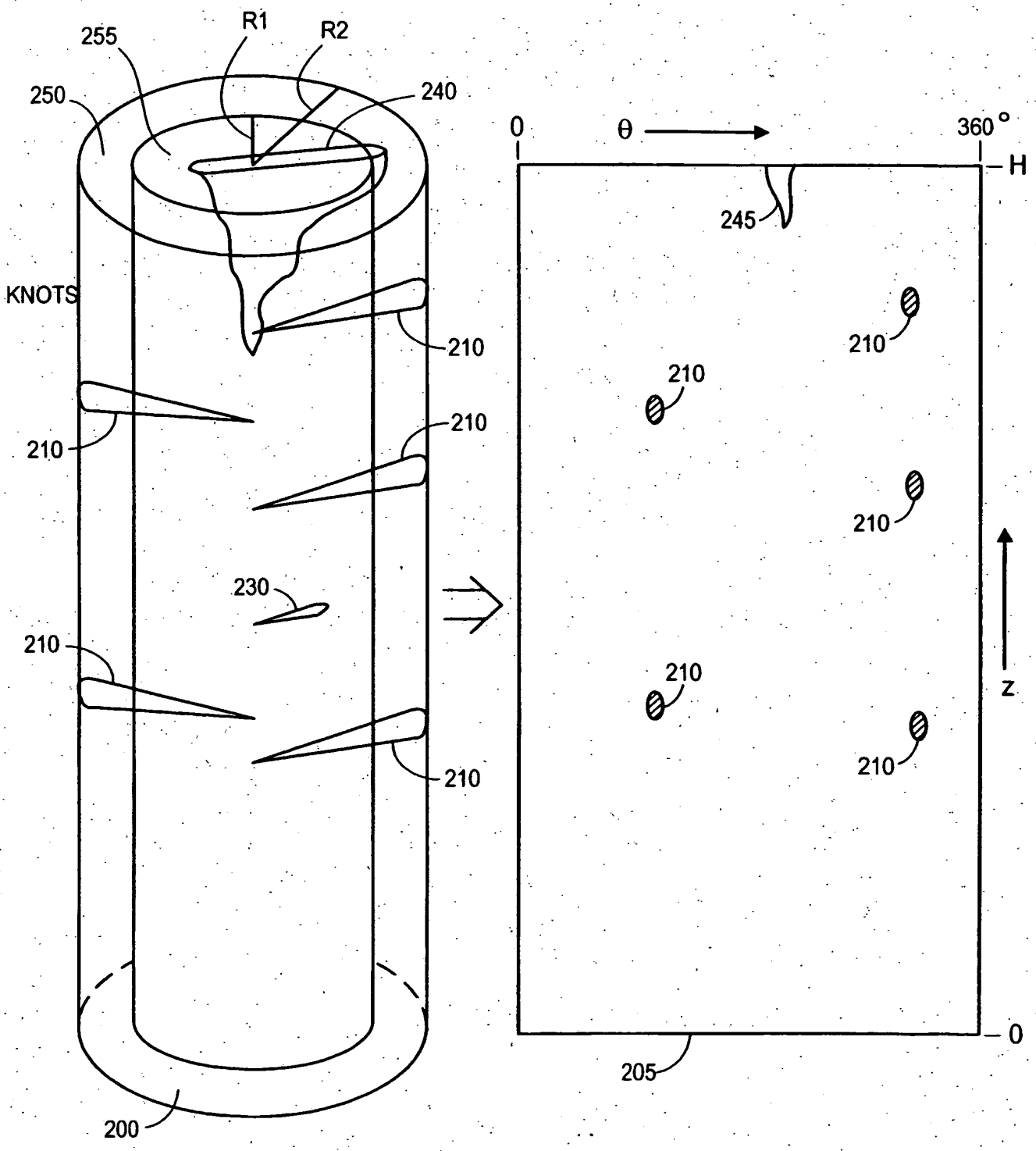

255
R1
R2
250
240
KNOTS
210
210
210
230
210
210
200
0
θ
360°
H
245
210
210
210
210
210
z
0
205

Fig. 2

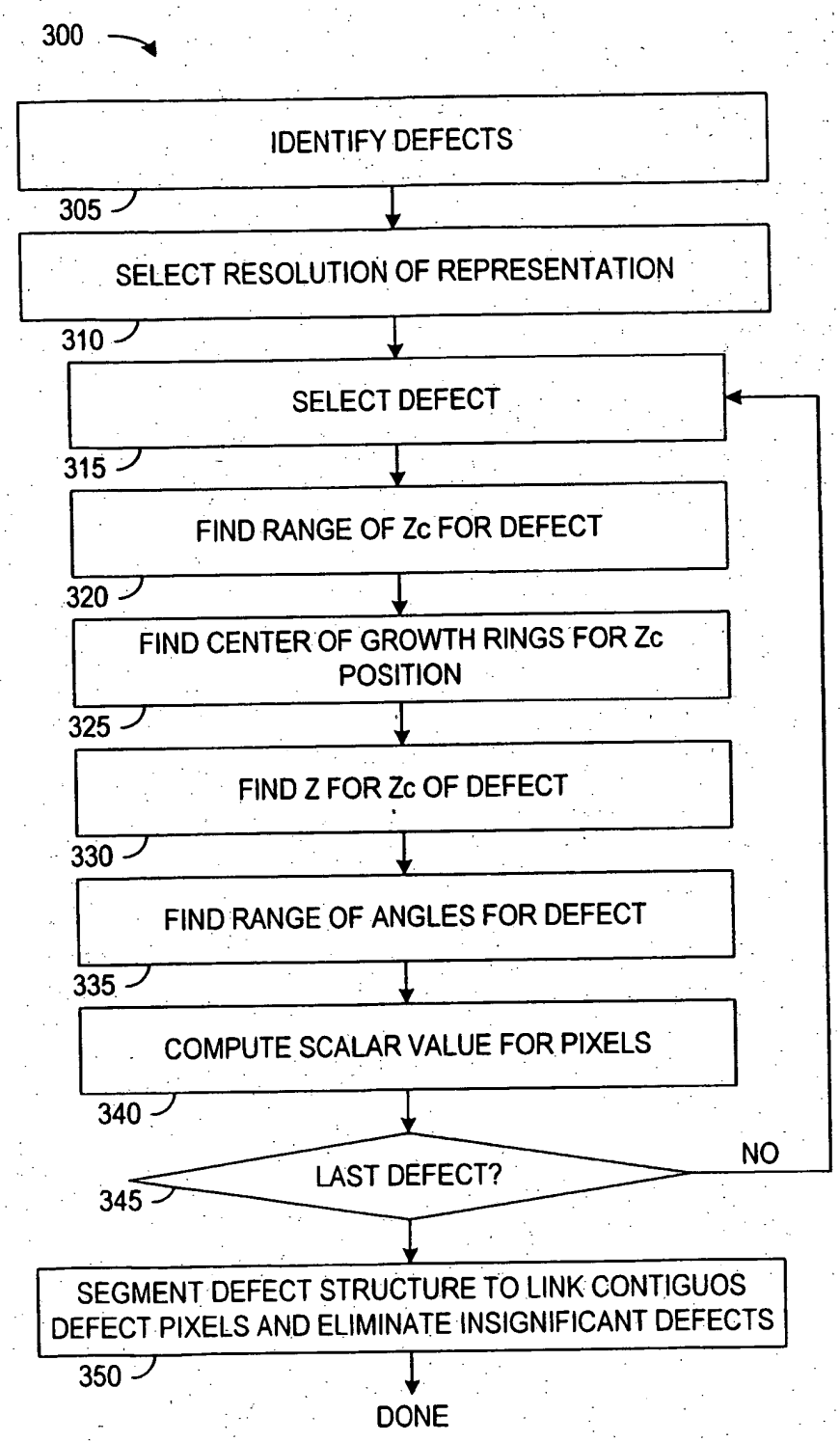

300
IDENTIFY DEFECTS
305
SELECT RESOLUTION OF REPRESENTATION
310
SELECT DEFECT
315
FIND RANGE OF Zc FOR DEFECT
320
FIND CENTER OF GROWTH RINGS FOR Zc POSITION
325
FIND Z FOR Zc OF DEFECT
330
FIND RANGE OF ANGLES FOR DEFECT
335
COMPUTE SCALAR VALUE FOR PIXELS
340
LAST DEFECT?
345
NO
SEGMENT DEFECT STRUCTURE TO LINK CONTIGUOS DEFECT PIXELS AND ELIMINATE INSIGNIFICANT DEFECTS
350
DONE

FIG. 3A

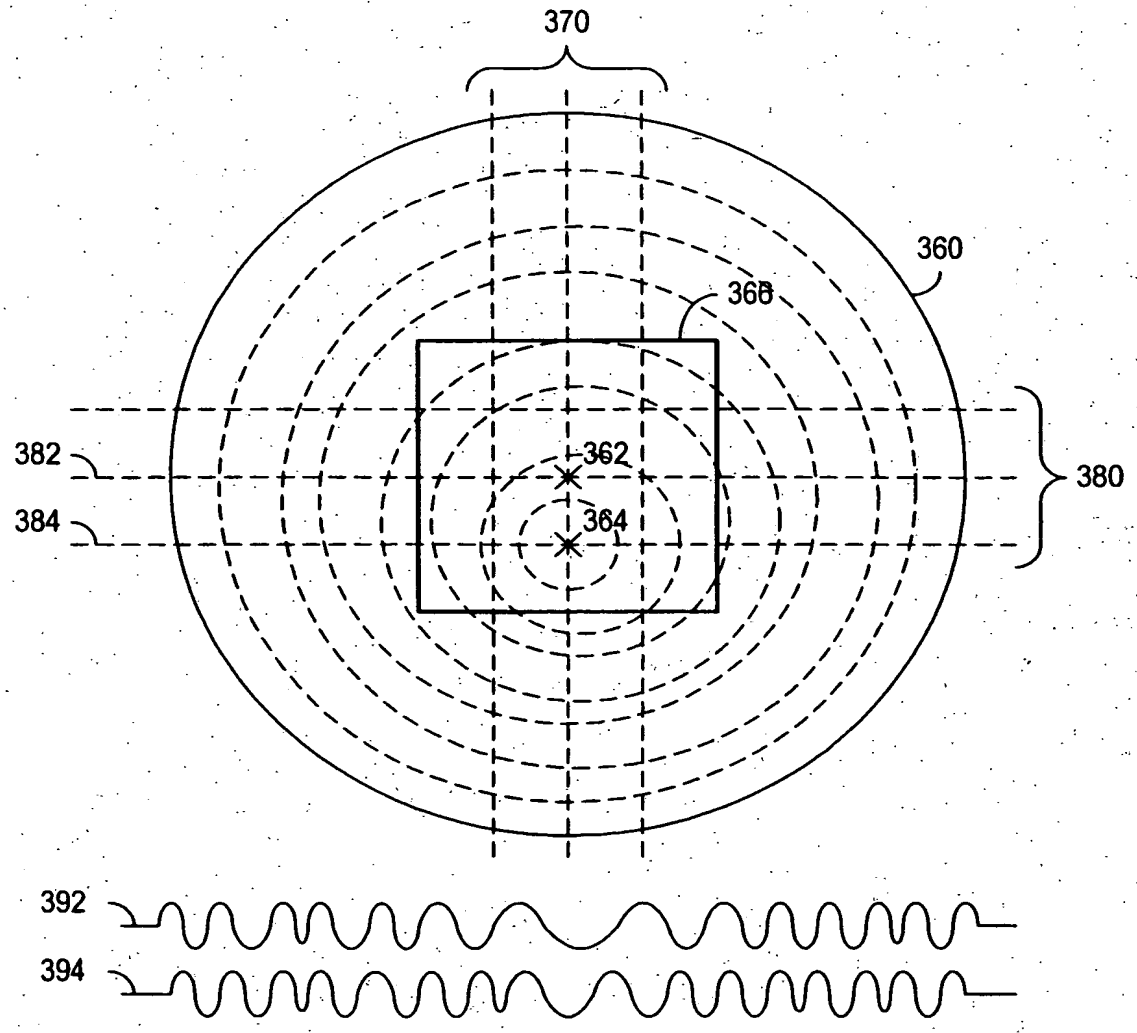


FIG. 3B

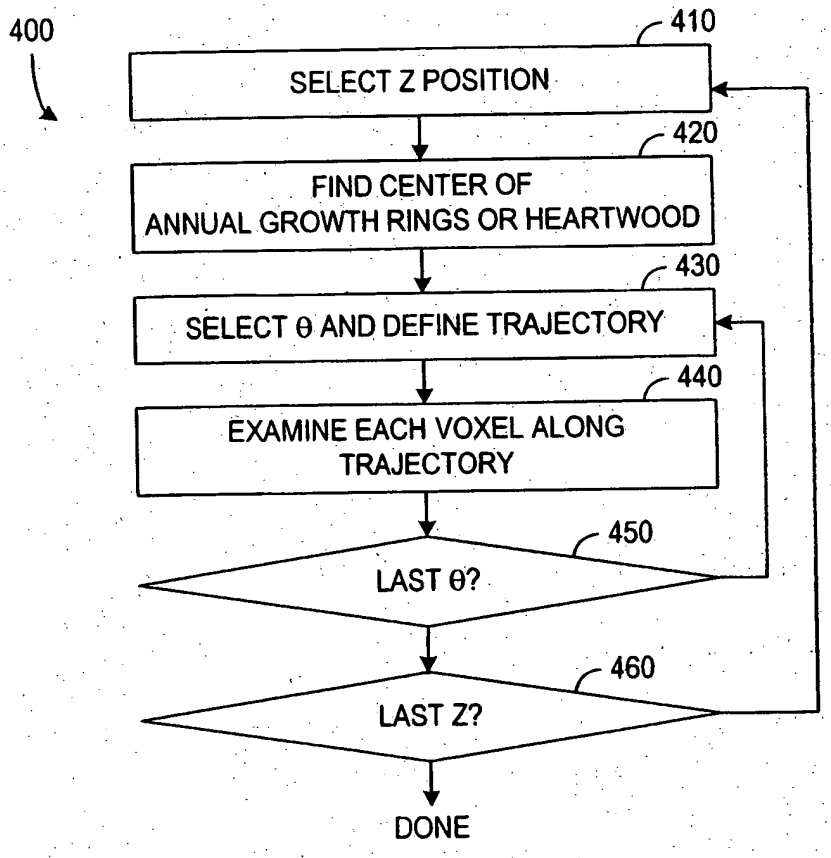

400
410
SELECT Z POSITION
420
FIND CENTER OF ANNUAL GROWTH RINGS OR HEARTWOOD
430
SELECT θ AND DEFINE TRAJECTORY
440
EXAMINE EACH VOXEL ALONG TRAJECTORY
450
LAST θ?
460
LAST Z?
DONE

FIG. 4

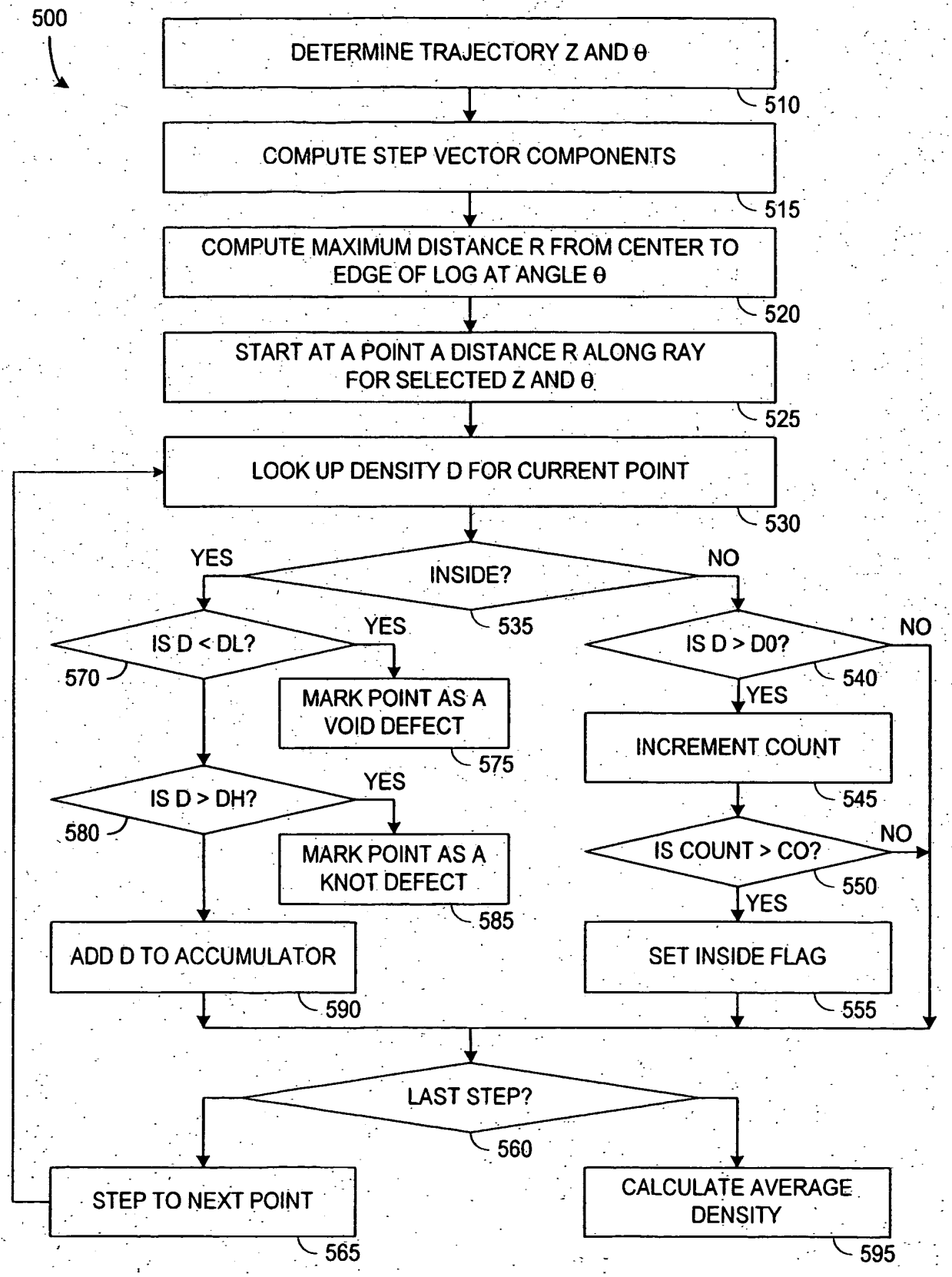

500
DETERMINE TRAJECTORY Z AND θ
510
COMPUTE STEP VECTOR COMPONENTS
515
COMPUTE MAXIMUM DISTANCE R FROM CENTER TO EDGE OF LOG AT ANGLE θ
520
START AT A POINT A DISTANCE R ALONG RAY FOR SELECTED Z AND θ
525
LOOK UP DENSITY D FOR CURRENT POINT
530
INSIDE?
535
YES
NO
IS D < DL?
570
YES
MARK POINT AS A VOID DEFECT
575
IS D > DH?
580
YES
MARK POINT AS A KNOT DEFECT
585
ADD D TO ACCUMULATOR
590
IS D > D0?
540
NO
YES
INCREMENT COUNT
545
IS COUNT > CO?
550
NO
YES
SET INSIDE FLAG
555
LAST STEP?
560
STEP TO NEXT POINT
565
CALCULATE AVERAGE DENSITY
595

FIG. 5

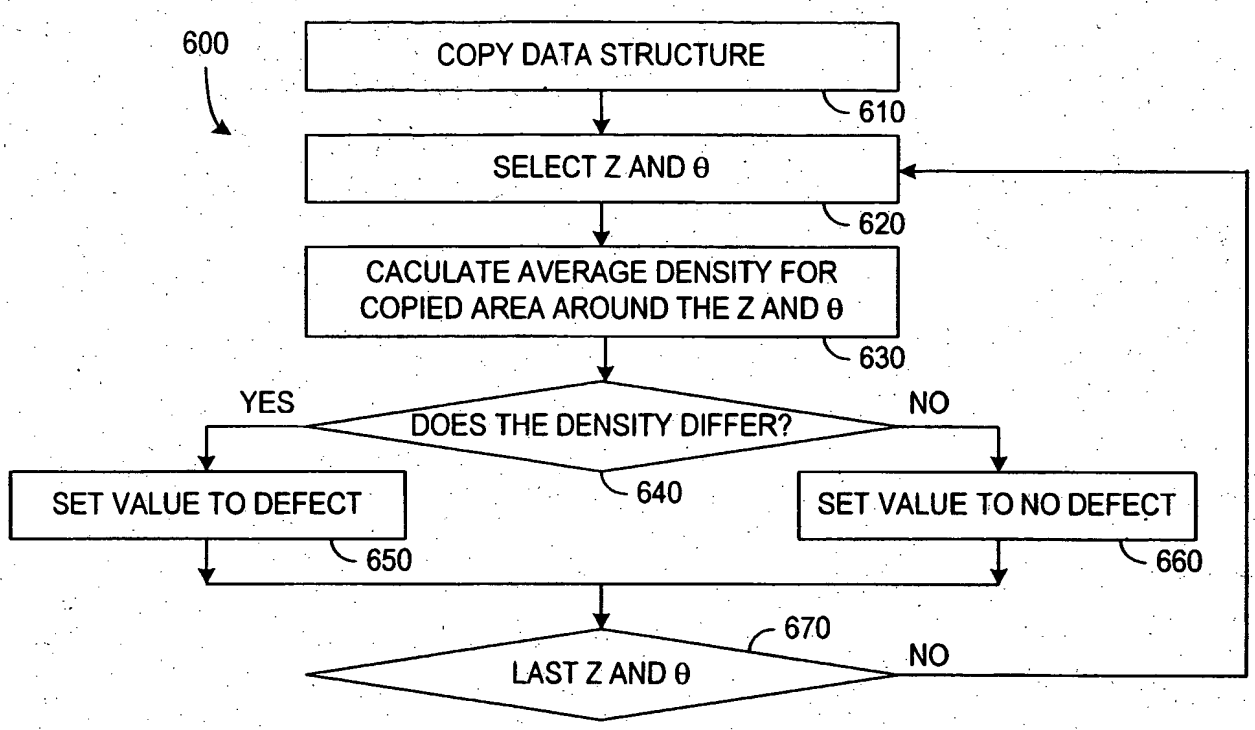

600
COPY DATA STRUCTURE
610
SELECT Z AND θ
620
CACULATE AVERAGE DENSITY FOR COPIED AREA AROUND THE Z AND θ
630
YES
DOES THE DENSITY DIFFER?
NO
640
SET VALUE TO DEFECT
650
SET VALUE TO NO DEFECT
660
670
LAST Z AND θ
NO

FIG. 6

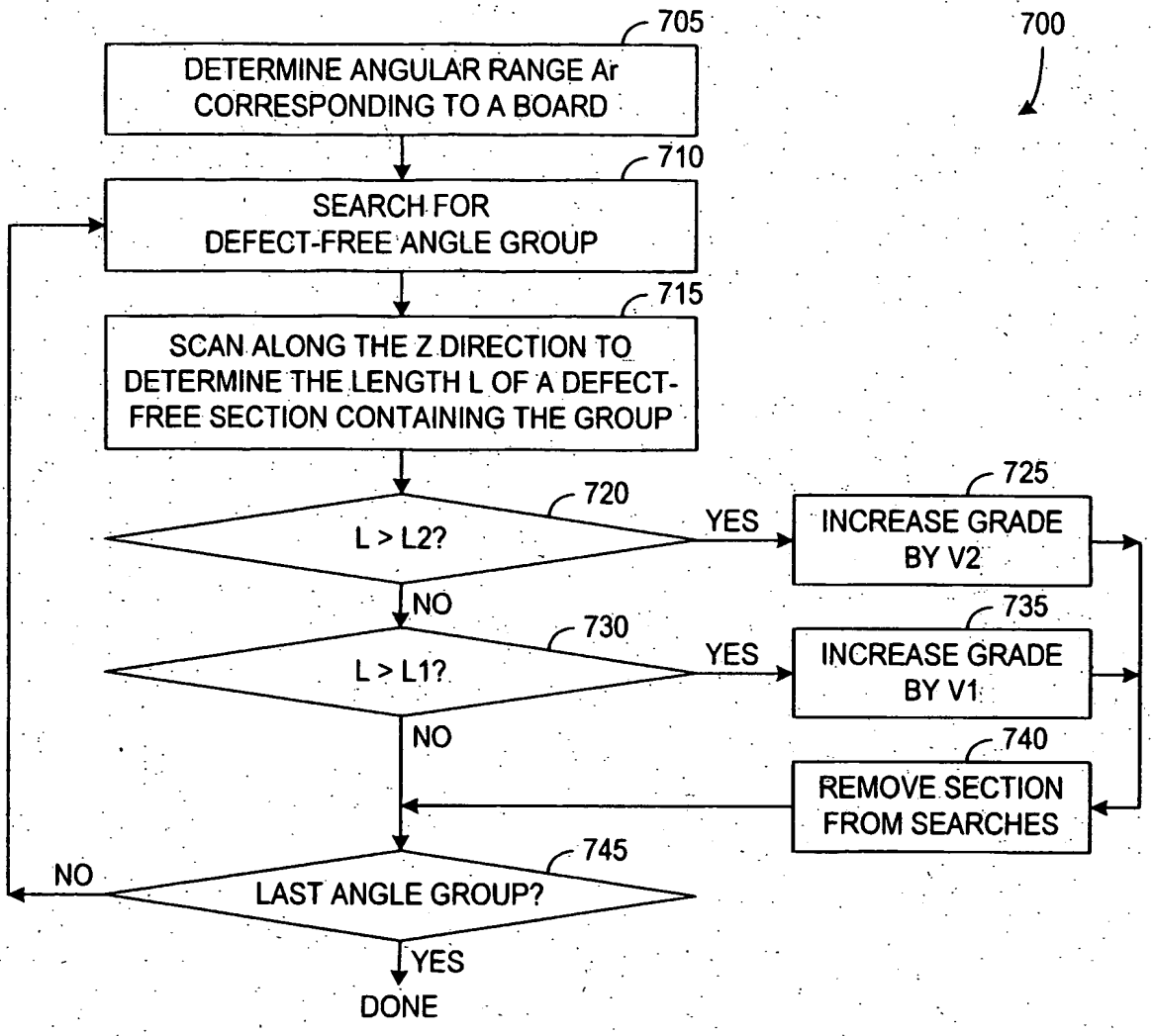

700
705
DETERMINE ANGULAR RANGE Ar CORRESPONDING TO A BOARD
710
SEARCH FOR DEFECT-FREE ANGLE GROUP
715
SCAN ALONG THE Z DIRECTION TO DETERMINE THE LENGTH L OF A DEFECT-FREE SECTION CONTAINING THE GROUP
720
L > L2?
YES
725
INCREASE GRADE BY V2
NO
730
L > L1?
YES
735
INCREASE GRADE BY V1
NO
740
REMOVE SECTION FROM SEARCHES
745
LAST ANGLE GROUP?
NO
YES
DONE

FIG. 7A

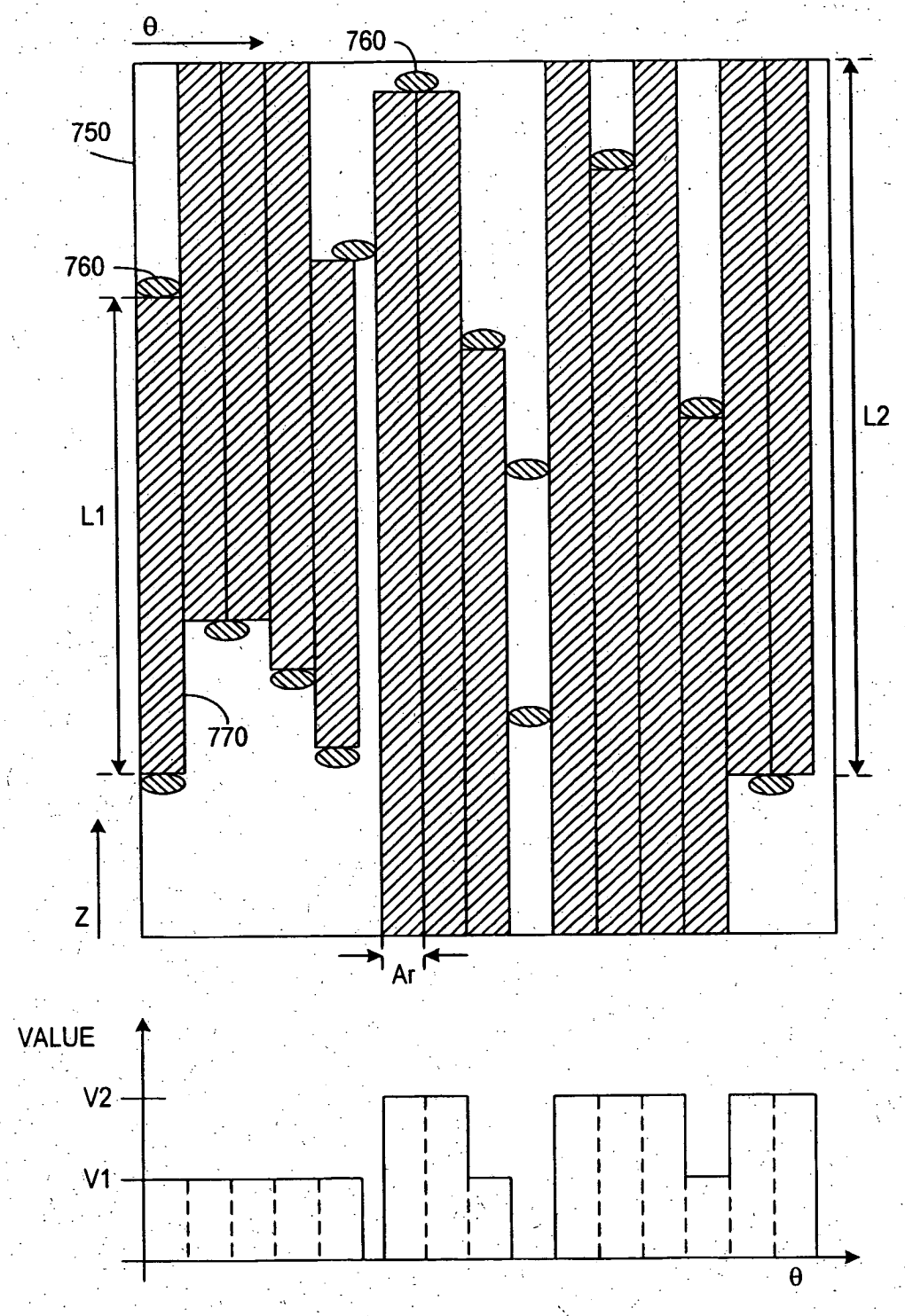


FIG. 7B

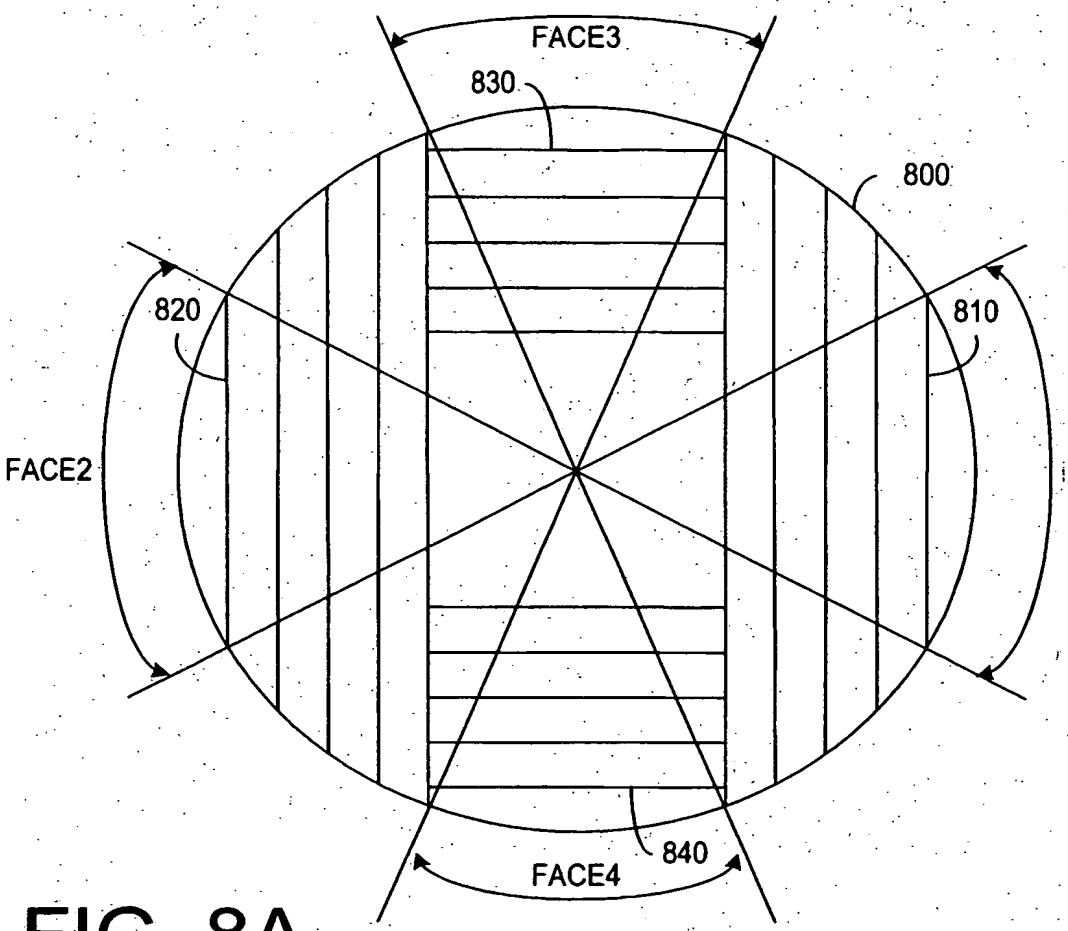

FACE3
830
800
820
810
FACE2
840
FACE4

FIG. 8A

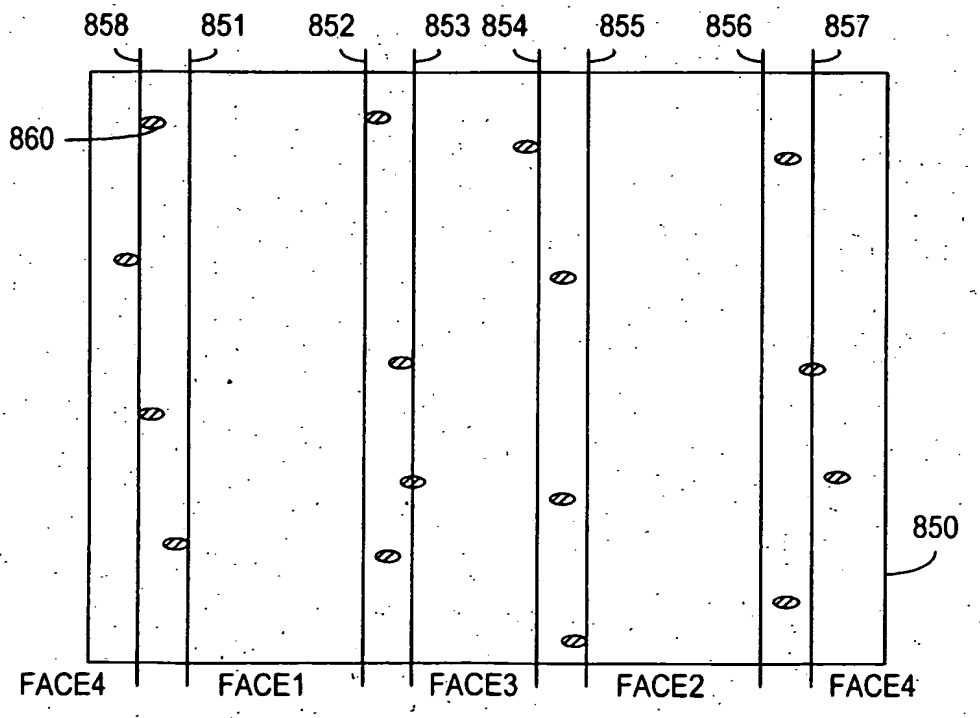

858
851
852
853
854
855
856
857
860
850
FACE4
FACE1
FACE3
FACE2
FACE4

FIG. 8B

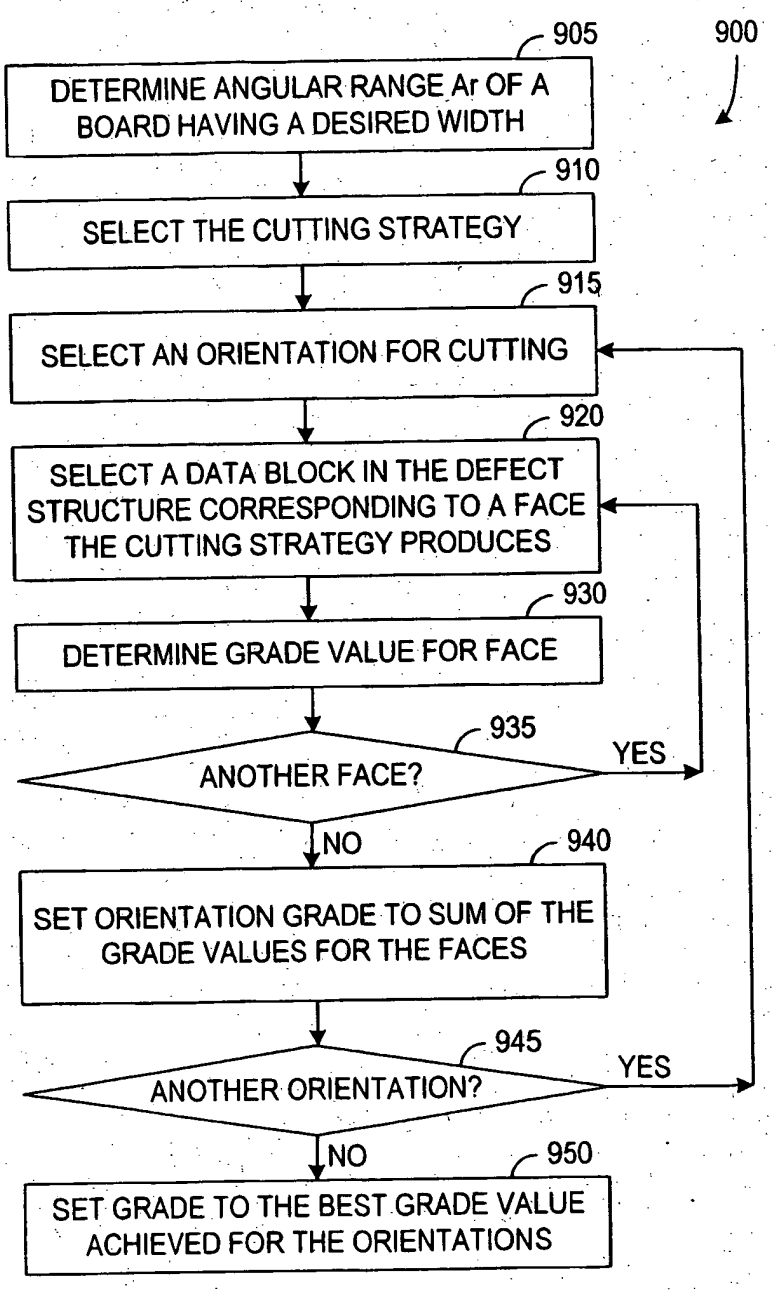

900
905
DETERMINE ANGULAR RANGE Ar OF A BOARD HAVING A DESIRED WIDTH
910
SELECT THE CUTTING STRATEGY
915
SELECT AN ORIENTATION FOR CUTTING
920
SELECT A DATA BLOCK IN THE DEFECT STRUCTURE CORRESPONDING TO A FACE THE CUTTING STRATEGY PRODUCES
930
DETERMINE GRADE VALUE FOR FACE
935
ANOTHER FACE?
YES
NO
940
SET ORIENTATION GRADE TO SUM OF THE GRADE VALUES FOR THE FACES
945
ANOTHER ORIENTATION?
YES
NO
950
SET GRADE TO THE BEST GRADE VALUE ACHIEVED FOR THE ORIENTATIONS

FIG. 9

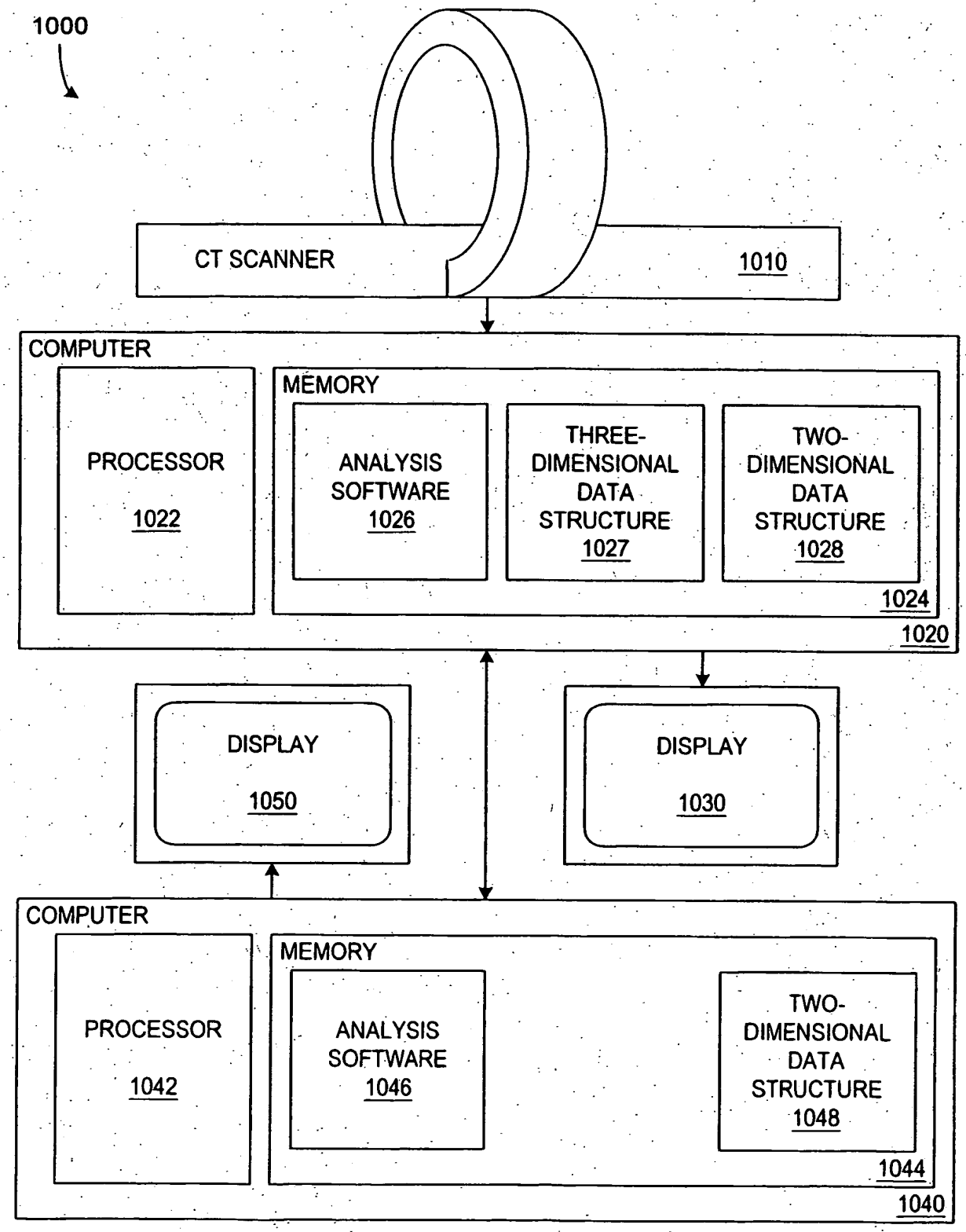

1000
CT SCANNER
1010
COMPUTER
PROCESSOR
1022
MEMORY
ANALYSIS SOFTWARE
1026
THREE-DIMENSIONAL DATA STRUCTURE
1027
TWO-DIMENSIONAL DATA STRUCTURE
1028
1024
1020
DISPLAY
1050
DISPLAY
1030
COMPUTER
PROCESSOR
1042
MEMORY
ANALYSIS SOFTWARE
1046
TWO-DIMENSIONAL DATA STRUCTURE
1048
1044
1040

FIG. 10